DYNAMIC NETWORK SECURITY CONTROL USING SOFTWARE DEFINED NETWORKING

THESIS

Michael C. Todd, Captain, USAF

AFIT-ENG-MS-16-M-049

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-MS-16-M-049

DYNAMIC NETWORK SECURITY CONTROL USING SOFTWARE DEFINED NETWORKING

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Cyber Operations

Michael C. Todd, B.S.C.S., M.S.C.S.

Captain, USAF

March 2016

AFIT-ENG-MS-16-M-049

DYNAMIC NETWORK SECURITY CONTROL USING SOFTWARE DEFINED NETWORKING

THESIS

Michael C. Todd, B.S.C.S., M.S.C.S.
Captain, USAF

Committee Membership:

Dr. Barry E. Mullins
Chair

Dr. Timothy H. Lacey
Member

Dr. Michael R. Grimaila
Member

AFIT-ENG-MS-16-M-049

Abstract

Software Defined Networking (SDN) introduced a new capability for programmers where network traffic could be managed unlike ever before. Whereas traditional network management solutions are rigid and limited to manual configurations, SDN allows for creative and adaptive solutions to common networking challenges. One of these key challenges is security, and specifically how to protect a network from attackers. This thesis develops and implements a process to rapidly respond to host level security events using SDN flow table updates, role-based flow classes, and Advanced Messaging Queuing Protocol to automatically update configuration of switching devices and block malicious traffic. By pushing security event information from each host to the controller and maintaining a security index for each host at the controller, the entire network is better protected from attackers.

A prototypical SDN-controlled network is constructed consisting of five hosts, one message broker, one controller, and one switch are used to measure system performance. Events are generated at each host to measure the Event Completion Time (ECT), Event Success Rate (ESR), and Message Delivery Time (MDT). These metrics are used to compare how the system performs as the number of events is varied between 1,000 to 50,000. Further analysis is performed to explain system response variations and make implementation recommendations.

Results show flow table updates are completed for all tested levels in less than 5.27 milliseconds and ECT increases with treatment level as expected. As the number of events increases from 1,000 to 50,000, the design scales logarithmically caused mainly by MDT. Event processing throughput is limited primarily by the message rate of the agent (40 messages/second). Additionally, the maximum effective consume rate for the controller indicates this design is capable of supporting up to 380 hosts at one message per second. Finally, every event initiated from all agents is successfully processed for both experiments resulting in a 100% ESR.

Acknowledgements

I am grateful to the faculty and staff of AFIT for providing an enlightening and challenging continuation of my life long journey of education. I am especially thankful to my advisor Dr. Barry E. Mullins and committee members Dr. Timothy H. Lacey and Dr. Michael R. Grimaila for their efforts to ensure the quality of this work. Most importantly I thank my family for understanding, loving, and thriving in the hectic world of military spouse and children.

Michael C. Todd

Table of Contents

Page

Page

Page

List of Figures

List of Tables

List of Abbreviations

Abbreviation		Page
DSCS	Dynamic Security Control Using SDN	22
JSON	Javascript String Object Notation	28
URI	Uniform Resource Identifier	38
MDT	Message Delivery Time	42
ECT	Event Completion Time	42
ESR	Event Success Rate	42

DYNAMIC NETWORK SECURITY CONTROL USING
SOFTWARE DEFINED NETWORKING

I. Introduction

This research implements a system to propagate host-level security countermeasures to switching devices using a host-based agent, Advanced Messaging Queuing Protocol (AMQP) message broker, Software Defined Networking (SDN) controller (Floodlight), and an OpenFlow enabled switch. This chapter introduces and provides context to the problem and the approach to address it. Next, the goals and expected results are presented as a reference point used to test the developed framework. Finally assumptions and limitations for this research are explained.

1.1 Background

The Information Age brought great advances in efficiency, productivity, and convenience in computing. Computing devices now support services spanning from basic utilities to military defense systems. While the level of security required depends on the information or resource being protected, host and network security is important to the success of most organizations. One approach is to apply host and network-based security systems, which typically come in the form of antivirus or intrusion detection/prevention products to manage these threats. However, since traditional networks require manual configuration, an antivirus alert does nothing to protect other hosts using the same infrastructure from the detected threat. A common reaction to detecting a malicious Internet Protocol (IP) address is to block it at border devices. This approach, while somewhat effective is not appropriate in all scenarios, especially those where services must be available.

Following the release of OpenFlow in 2007 [1], SDN enabled the separation of the control (configuration) and data (network traffic) planes reducing the requirement for manual

configuration of network devices. OpenFlow enables network switching devices to be configured though a controller. These controllers run custom applications created by programmers (e.g., Quality of Service, Security). Though some studies use network metadata to affect network configuration [2, 3], none directly test event detection and notification (using AMQP) on each host a to cause effects on the control plane.

1.2 Problem Statement

This research develops and implements a process to rapidly respond to host-level security events by using SDN flow table updates, role-based flow classes, and AMQP to update the configuration on switching devices and block malicious traffic. According to Symantec, 841 targeted attack campaigns were observed in 2014 [4]. Upon gaining access to a workstation, the attacker traditionally attempts to move laterally to other hosts on the same network. After establishing a foothold, the attacker finds a way to elevate privileges and steal data. Preventing an attacker from pivoting from one host to another slows the attack cycle and ultimately inhibits their capabilities. If an attacker gains access to fewer machines, the number of compromised user accounts is limited, thereby reducing the amount of data accessible by the attacker. Focusing on assessing if effective and efficient dynamic layer-2 management is possible, this research uses role-based flow profiles and existing security software to manage layer-2 flow tables. Ultimately, this research aims to address this problem and reduce the effectiveness of attackers by inhibiting their mobility and persistence throughout a network.

1.3 Goals and Hypothesis

The objectives of this thesis are to develop and evaluate a process for managing network traffic from hosts using an SDN controller and flowtable updates. The goal of this research is to implement a system for near real time control of flow tables without inadvertently denying critical communication channels. Thereby reducing the possibility of an unfavorable outcome (e.g, loss of data). The components used are flow table updates to OpenFlow-

enabled layer-2 devices, one modified Floodlight Stateless Firewall application, one AMQP message broker and an agent at each host.

It is hypothesized that as the number of security incidents (events) sent to the controller increase, the time to complete each event will increase but remain lower than one second, the message delivery time will stay relatively static, and the event success rate will remain at 100%. By ensuring network flows are within the expected values and blocking or dropping unexpected data at the switch, unauthorized data transmission can be stopped at the first hop. The time interval of one second is used based on the assumption that the time required for an attacker to gain access and steal data from a victim is greater than one second.

1.4 Approach

A network consisting of five nodes, one switch, one SDN controller, and one AMQP message broker is deployed. Time is monitored throughout the process at the host agent, message broker, and SDN application (before and after the event is complete). These time measurements are used to calculate the total time it takes to deliver a message to the SDN controller and to fully process each event.

1.5 Assumptions and Limitations

The experiments use some of the most common security events observed on the Air Force Network (AFNet). The list of cases is compiled through coordination with the Air Force Computer Emergency Response Team (AFCERT). This list includes malware detected, unauthorized media detected, authentication failure, and registry modification or tampering.

1.5.1 Security Event Detection.

Detecting a security event is not the focus of this research and is abstracted away to the aforementioned event types randomly generated during each experiment. Research is limited to how fast the system can make desired changes to configuration and the ease of

adaptation to existing networks.

1.6 Contributions

This thesis contributes to the SDN body of research. Specific contributions include a process for efficiently managing layer-2 flow table updates and programming the control plane considering the security state of each end node.

1.7 Thesis Overview

This thesis is organized into six chapters. Chapter 2 defines security, SDN, and presents relevant research needed to understand the proposed process being developed. Chapter 3 presents the approach to the problem, a detailed explanation of how each component works, and key decisions during development. The methodology for evaluating the process using SDN capabilities is covered in Chapter 4. The results for each experiment and analysis of collected data are presented in Chapter 5. Finally, Chapter 6 summarizes the research and concludes with an explanation of the significance of this research and suggested future work.

II. Background and Related Research

This chapter defines network and host security and discusses the traditional bodies of research employed to secure a network. Section 2.1 introduces and defines the principles of network and host-based security techniques and the concept of a holistic approach to enterprise security. A thorough review of SDN technologies are presented by introducing the control and data planes, OpenFlow, and SDN controllers in Section 2.2. In Section 2.3 reliable messaging is discussed, specifically the Open Internet standard AMQP. Section 2.4 introduces the principles of Role-based Access Control. This chapter concludes with a description of research ideas used for this thesis in the context of a process to use SDN with existing host and network security technologies.

2.1 Security Principles

Enterprise security concerns all devices from end hosts to intermediate switches and routers. Traditional enterprise security is logically split into two classes, network and host security. The network class includes security zones, access control, appliances such as a firewall or Intrusion Detection System (IDS), and encrypted channels. Host or node based security includes policy models implemented using access control, permissions, software updates, and authentication schemes. Additional measures include registry or kernel monitoring and software such as antivirus, IDSs, or application security like Secure Socket Layer (SSL) [5]. While it is outside the scope of this research to explain these classes in detail, two observations are important to note. First, many different technologies have been developed in each of these two classes and in some cases exist in both (e.g., authentication and intrusion detection). Second, few of these techniques propagate awareness (events) to influence both classes concurrently. For example, many networks are customized with a series of devices to perform signature-based match-action events. Before SDN these systems lacked the flexibility, control, and scalability necessary to manage complex networks in real time without considerable equipment and resources. A secure network model includes the following [6, 7]:

- Access - those authorized can communicate to and from the network
- Confidentiality - information is protected while in transit
- Authentication - users are verified to be who they claim to be
- Integrity - the message being sent was not modified during transit
- Non-repudiation - each user can be decisively held accountable for their use of the network

The following sections discuss the association of risk and security and the current technologies employed to maintain access, confidentiality, authentication, integrity and non-repudiation.

2.1.1 Security and Risk.

Security is discussed by many as a risk management optimization problem; given all feasible solutions, pick the best (least costly). A good example of this concept is the software risk management framework. According to Dowling *et al.*, Risk Exposure (R(E)) [8] is (1):

$$R(E) = P(OU) \times L(OU) \tag{1}$$

Where Probability of Unsatisfactory Outcome is $P(OU)$ and Loss Given Unsatisfactory Outcome is $L(OU)$. This system depends on risk assessment and control to gather and evaluate pertinent details to choose the best course of action. Naturally, decision makers need to know as many relevant details as possible to make the best decision.

In the case of a computer network, a prime source of information is the end host. In this approach, hosts are scanned at a set interval and system state is monitored. Unlike many attempts at security, when leveraging SDN, change to security configuration does not have to stop at the host. A goal of this research is to mitigate $R(E)$ by presenting a low cost network security framework that also lowers the probability of unsatisfactory outcome. While network traffic is also a key source of security information, only host data is considered in this research.

2.1.2 Security Incidents.

Security incidents (with malicious intent), referred to as events in this research, occur when an attacker with some objective uses tools to access unauthorized resources through a discovered vulnerability by performing actions on targets [9]. Actions such as scan, bypass, or steal may be focused on targets like a network, component (e.g., firewall), or data respectively. This research focuses on responding to an event in the sense of an action focused on a target. In Figure 1, Howard and Longstaff define an incident to contain one or more attacks where each attack contains one or more events [9]. The events used in this research are defined in this manner and classified in Table 1 in Section 3.3.1.1.

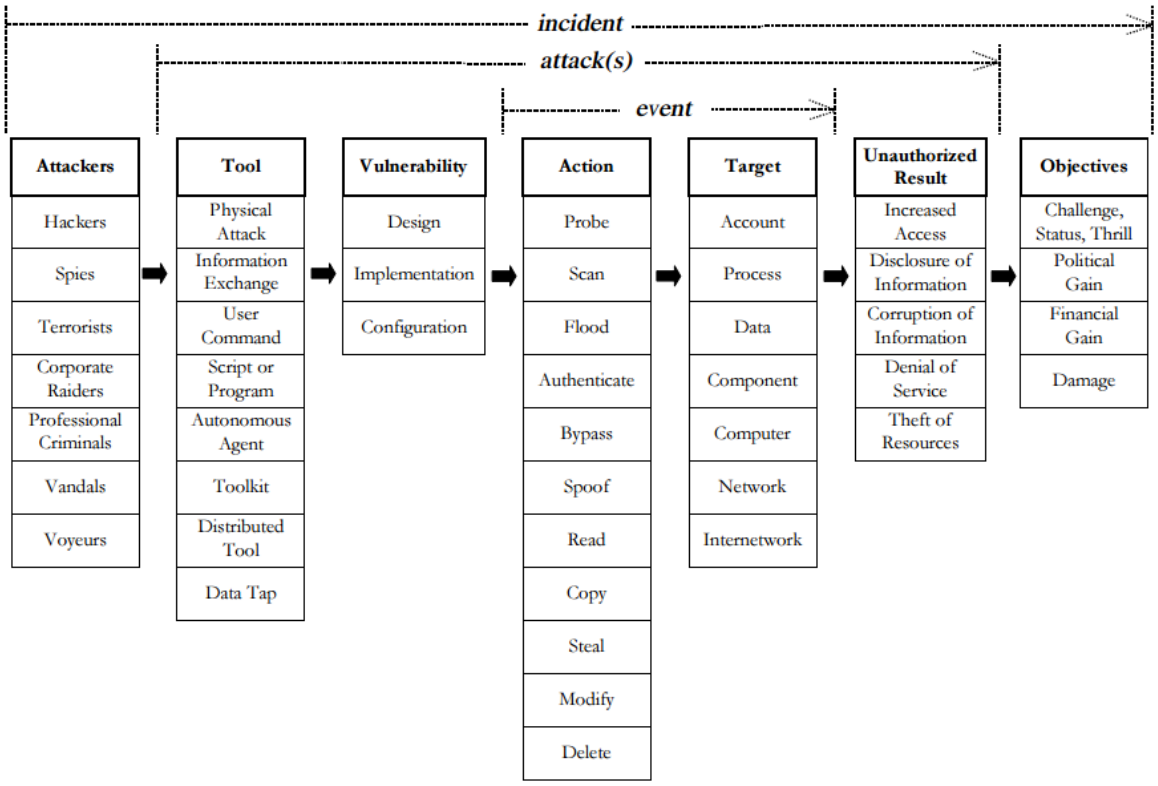


Figure 1. Security Incident Framework [9]

2.1.3 Holistic Security Approach.

It is evident, especially in military organizations, a unit is only as secure as its weakest member. In the world of computing and networks, this means each device (and host) must be protected. Many products attempt to protect a single device, such as an antivirus suite, but this does not protect other hosts from being targeted from the same avenue of attack. Other systems focus on detecting malicious network traffic using middle boxes like an IDS or firewall. Neither solution is adequate to ensure a secure enterprise; however, the combination of both can bolster security posture while reducing risk within the enterprise. The USAF currently uses a system known as Host Based Security System (HBSS) which monitors, detects, and counters known cyber-threats [10]. HBSS is a Commercial Off The Shelf (COTS) product based on McAfee's Enterprise ePolicy Orchestrator (ePO) suite and simplifies deployment of systems, security policies, and reduces infrastructure overhead. ePO claims to synthesize global threat intelligence along with endpoint and network security and compliance [10]. As attackers' tactics, techniques, and procedures evolve, holistic security technologies such as HBSS are a necessity to ensure access, confidentiality, authentication, integrity, and non-repudiation are safeguarded. While HBSS does reach all devices across the enterprise, policy updates and system monitoring does not affect layer-2 switching rules. By coupling existing technologies such as HBSS, SDN, and AMQP, greater network situational awareness and security are achievable.

2.1.4 Network-based Security.

Network-based security schemes aim to secure resources inside a protected network. Networks are separated into physical and logical zones. These zones are commonly described as public facing Demilitarized Zone (DMZ) (internal unprotected) and private intranet (internal protected) [5]. Within and on the edge of each zone, network appliances perform filtering, forwarding, inspection, and accounting. Firewalls filter or forward traffic using Access Control Lists (ACL) and a match-action operation. Inspection devices such as an IDS or Intrusion Prevention Systems (IPS) rely on signatures or anomaly detection to

determine if malicious traffic is present and in the case of the IPS, update rules or policies to prevent said activity [11]. Unless tunneled through a secure channel like a Virtual Private Network (VPN), secure data transfer is normally accomplished by applications on hosts using SSL or Transport Layer Security (TLS) [5].

2.1.4.1 Switches and Routers.

Switches and routers forward information from one port or interface to another. Speed is imperative to ensure data is transfered from source to destination as fast as possible. These devices are managed by network administrators and do not keep record (logs) of communications beyond hardware and network addresses [12]. This limitation is functional since logging is computationally expensive and storage intensive. These devices process traffic at near real time (microseconds) and with the exception of customized solutions are not updated based on the changing state of end hosts. To achieve dynamic control of device configuration and still maintain real time speed, this research leverages OpenFlow-capable switches and the programmability of the network plane using SDN applications as described in Section 2.2.

2.1.4.2 Network-based Detection and Prevention.

Figure 2 presents results from the annual Internet Security Threat Report published by Symantec, where identification of zero day malware increased by 300% from 2011 to 2014 [4]. Network IDS/IPS are useful to identify malicious activity on hosts based on the traffic they generate. The three main tools these systems provide to incident responders is prevention, detection, and mitigation. These devices record incident information, notify a human or system, and distill raw data into meaningful information for reporting [13]. The common detection methodologies include signature-based, anomaly-based, and stateful protocol analysis and can be implemented in the network or host environment [14].

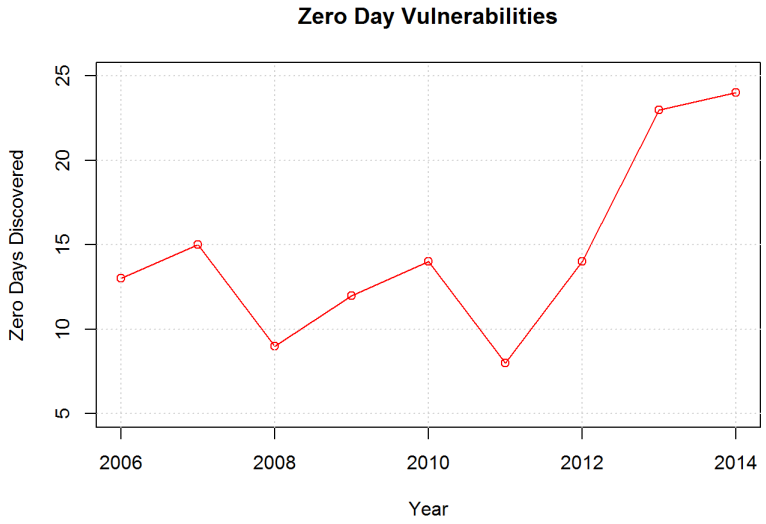


Figure 2. Symantec Threat Report 2015 - Zero Day Statistics [4]

Signature detection systems are effective but do not reveal zero day exploits, especially those encrypted and disguised as legitimate traffic. Comparing current activity to a definition of *normal* is the approach behind anomaly-based detection. Stateful analysis is also effective but requires space, processing power, and time to implement. Because of these requirements, stateful protocol analysis does not scale as well as signature based. Two open source IDS solutions for signature and stateful protocol analysis are snort and Bro respectively [15, 16]. Though the framework implemented in this research does not use a network IDS, the flexible messaging system and status framework employed allows for integration with any platform with system state information and Java, Python, or C/C++.

2.1.5 Host-based Security.

Host based IDS/IPS run as agents on a single host and monitor the state of a system using various algorithms. Some of the characteristics monitored include network traffic, logs, processes, files attributes, and system configuration [14]. The agent is highly customizable and configured based on what it is designed to protect. One agent might be focused on the Operating System (OS), while another focuses on application monitoring. If an alert is

triggered, agents might communicate this information to a centralized server for reporting or take action to secure the host immediately. Since most hosts use Java/Python/C++, arguably any agent can be ported to the proposed framework using AMQP. A host-based agent analogous to HBSS is used for this research and simulates events detected from antivirus, access control, registry settings, or anomalies to determine the security state of each host (Section 3.1). This implementation is similar to HBSS and can be applied by the USAF if SDN-capable hardware and software are enabled/configured and HBSS agents updated.

While HBSS agents can detect and intercept malicious attempts on the host, they do not affect configuration change on network switching devices. This is because switching hardware is designed to execute basic match-action instructions. Awareness of hosts is limited to a MAC address and physical port assignment unless additional rules are configured such as a Virtual Local Area Network (VLAN).

2.1.6 Role-based Access Control.

Role-based Access Control (RBAC) is a non-discretionary security model. RBAC poses access to objects as a user to member-of group relationship. This construct results in a set of rules to govern access to objects based on a group's function [17]. Flat RBAC is the simplest implementation only requiring roles with a set of permissions and users assigned roles [18, 19]. This research applies RBAC principles to create a flow table profile for each role (workstation, server, etc.). Accordingly, each host on the network is assigned a role, managed by an SDN application, and allowed to access other hosts (internal or external) depending on the flow rules propagated using OpenFlow to switching devices. Section 3.1.2 explains the structure and function of the RBAC flow profile in detail.

2.2 Software Defined Networking

Software Defined Networking (SDN) is the use of software to manage packet forwarding devices, essentially taking strict control of transmission and other flow functions away from network devices. While this generalization seems simple, the application of SDN is far more

dynamic and continues to evolve. This section explains why this field is so dynamic and why allowing software to complement network flow control is so significant.

2.2.1 Network Control Evolution.

Computer networks traditionally rely on network switches, routers, and modems to provide connectivity to other computers. Data from one computer to its destination is managed by these devices and their employment of the transport and network interface layers as described in the Transmission Control Protocol/Internet Protocol (TCP/IP) request for comments (RFC) 793 [20]. The use of match-action devices (e.g., switch) is still the standard for packet forwarding in networking and is not changed by the development of SDN. Commonly referred to as decoupling the data and control plane, SDN enables software developers access to manage with particularity the previously inaccessible packet-forwarding mechanism. As proposed in [21] an example of leveraging this capability is to declare Quality of Service (QoS) requirements and for the underlying network to allocate resources on demand. Using an application programming interface (API) the programmer could signal the controller as bandwidth or response-time minimums are required, triggering potential configuration changes issued by the controller to the underlying virtual or physical switching infrastructure.

In 2007, a research team at Stanford University envisioned a network without the limitations of traditional hardware switching technologies. As computing and networks continue to grow, network infrastructure must scale to handle the load. In an effort to facilitate research and curb the cost of equipment and management, Martin Casado developed the modern concept of SDN and the open-source southbound API OpenFlow [1, 22]. The efforts of his team led to what is known as SDN.

2.2.2 The Control and Data Planes.

The purpose behind developing SDN was to make a traditionally rigid environment (a network) more flexible. In [22] Casado *et al.* describes the ideal "control plane" (layer) as being flexible, simple, inexpensive and vendor neutral. They also imply the ideal "data

plane" as including intelligent devices with virtual switching features built-in. It is hard to imagine a traditional network that meets any of these goals. In efforts to reach these goals, network devices and vendors thereof, must surrender proprietary control and just transfer data, leaving the management of traffic to a software controller. Figure 3 illustrates how the infrastructure for a network becomes two pronged, (i) physical hardware and (ii) software that controls it–hence the data and control layers [23]. The application layer is the final piece to the puzzle, where programmers use an API to interact with the control layer. In Figure 3 these three layers are formally described as the Application Layer, Control Layer, and Infrastructure Layer.

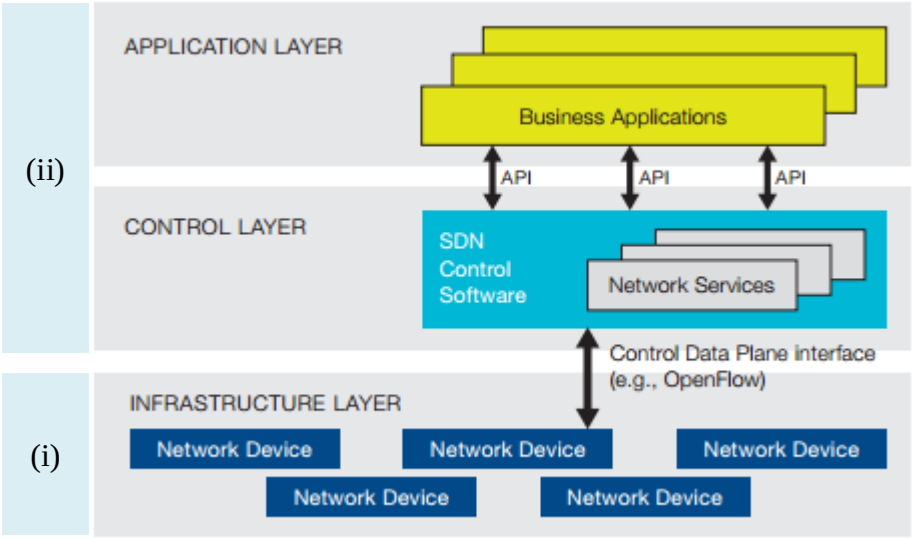


Figure 3. SDN Framework [23]

Between the application and control layers, developers are empowered with an API known as the Northbound API. This is where the control plane is affected by the application (or user), enabling calls to the control layer where the SDN control software or "Controller" is located. There, the controller provides network functions such as routing and security. Below the control layer is the infrastructure layer, where physical and virtual network devices reside. Network devices (e.g., switch) supporting OpenFlow reside at

the infrastructure layer and represent the data plane. This is where the control and data plane interface and the OpenFlow protocol is used to control devices. Presently no industry standard exists for SDN controllers, however, BigSwitch Networks and VMWare developed one vendor-agnostic API named Quantum that is employed in the OpenStack cloud service framework [24, 25].

2.2.3 Virtual Switching.

Virtual switching refers to delivering network traffic from one host to another (possibly without leaving the host operating system) using software or a combination of hardware and software. Without the proliferation of Virtual Machines (VMs) and affordable cloud computing solutions, SDN would arguably not be as popular as it is today. As hardware becomes more robust and software is developed to take advantage of these systems, a few high-performance servers might replace what was previously tens (eventually hundreds) of servers. With free software enabling virtualization like Oracles VirtualBox, using VMs on individual workstations is common place, especially in academia and research and development.

With virtual machines came the issue of network access. In early VMs, bridging virtual network devices to physical devices was common. As hardware and software advanced, whole networks could exist within a single physical machine. For example, Linux releases since 3.3 are vSwitch compatible and work seamlessly with VMWare and VirtualBox [26]. There are several virtual switching technologies such as Open vSwitch (OVS) [26], OpenFlow [22], Cisco Nexus 5000V [27], and IBM 5000V [28].

2.2.4 OpenFlow.

The Open Networking Foundation (ONF) is a non-profit open-source community charged with developing and accelerating the adoption of SDN [29]. ONF has a membership of over 100 companies with more than 64 OpenFlow products on the market. Since 2009, ONF released four major revisions to OpenFlow and the latest proposed specification (OpenFlow 1.5) was released in December of 2014. OpenFlow is the first official southbound API for

SDN to provide device management and configuration across several vendors. An OpenFlow switch consists of three main components–flow tables, a group table, and a channel. The channel is used to manage and communicate between an SDN controller and a switch. While the group and flow tables contain match-action rules (e.g., *match*: IP address p, *action*: forward to port q). As seen in Figure 4, as each flow is processed through the flow table pipeline, it is either matched or passed to the next table. If no match is found, the flow is processed with the configured default rule (e.g., forward, drop, or flood).

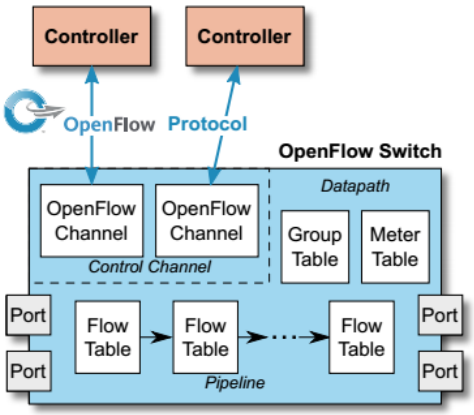


Figure 4. Components of an OpenFlow Switch [23]

The divergence from traditional switching is evident as there is a controller exterior to the switch managing flow tables. In traditional switching, flow rules are statically set by an administrator. Changing rules requires authenticating, loading, modifying, and saving the configuration–all introducing time, personnel cost, and potential human error. Pure SDN and hybrid switches working in OpenFlow mode process flows using the OpenFlow pipeline. This pipeline implies each packet is processed by each flow table and passed to the next if not matched. Each flow table is numbered starting at zero and increasing. As shown in Figure 5, if the flow table matches the packet, the instructions for that flow table are executed.

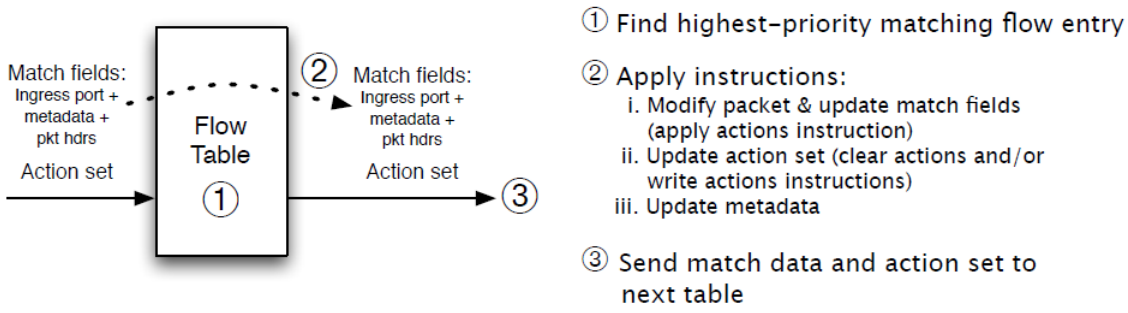


Figure 5. OpenFlow packet flow through the processing pipeline [23]

2.2.5 SDN Controllers and Programability.

Programming on the control plane, while being a new practice, was not a new concept when the Stanford team began building the OpenFlow standard. Starting in 1995, those devoted to the potential usefulness of a programmable network interface (or network API), first developed Active Networks and Control-Data Separation. By 2007 NOX/POX (the first OF controller) was released to the public and provided a much desired leap forward for SDN research [30, 31]. Given virtual switching and how the control and data plane are decoupled using SDN, the following sections describe the controller, their potential impact on a network, and popular publicly-available controllers–NOX, Floodlight, and Ryu (ree-you).

An SDN controller at the core is a software approach to manage network resources and provide an interface between network traffic and applications. For example, a custom application to handle layer-2 switching, could monitor and meter types of flows for accounting or security purposes. Any middle box solution such as IDS, firewall, or even network services like Domain Name Service (DNS) can be implemented as an application on top of a controller. Management of functions provided by traditional infrastructure is abstracted away to control software (applications).

2.2.6 Beacon and Floodlight.

Following NOX/POX, the next leap forward was the release of the Java based open source controller Beacon. Being integrated into the well known open source IDE Eclipse, made SDN applications available to any programmer from beginner to skilled professional. As a fork of the Beacon source, Big Switch Networks created their own version titled Floodlight. Built using the popular Apache Ant framework, Floodlight became even more accessible and fostered a very active community of SDN developers [32]. Figure 6 shows the Floodlight framework is structured so modular applications access core services through the Java API. The programmer has the freedom to innovate by developing modules (applications) such as forwarding, firewall, or learning switch. All of which is accessible using REST applications through the REST API.

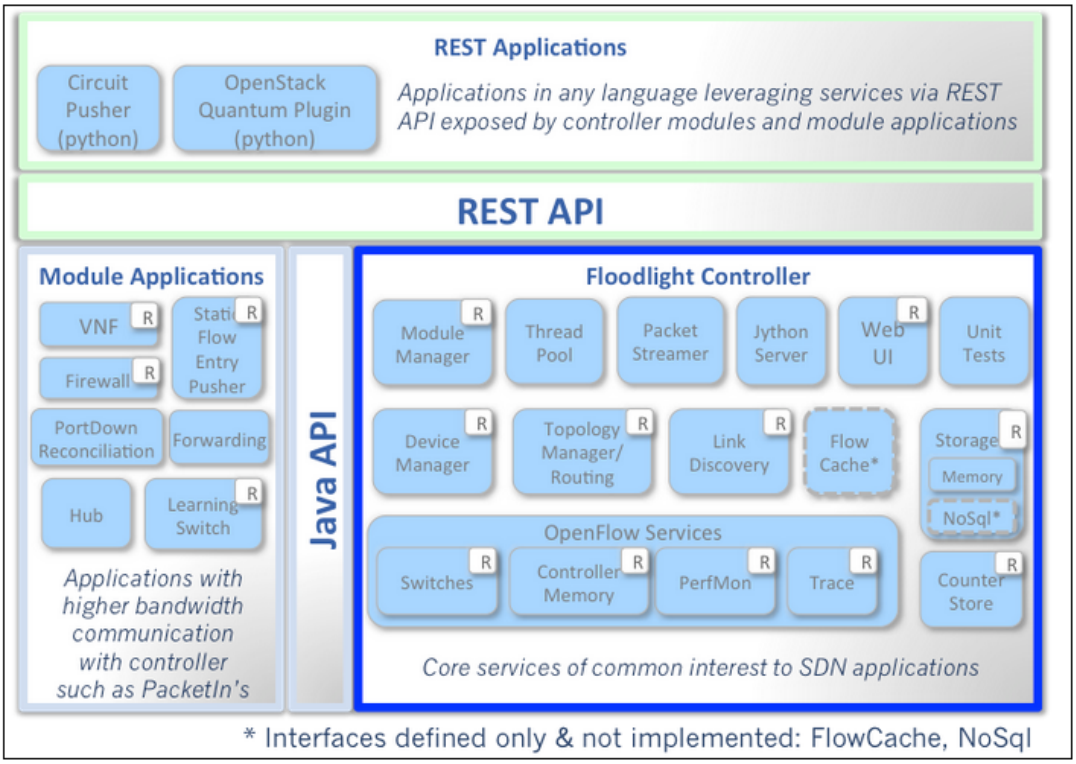


Figure 6. Floodlight Controller Framework [32]

Floodlight is used by a number of projects to enable virtual network and machine management. One of the flexible features of Floodlight is a Representational State Transfer

(REST) API which allows access to both applications and controller modules. One of the most popular open source projects to use Floodlight is OpenStack. Using the OpenStack Horizon GUI, virtual networks are managed along side VMs using the Floodlight REST API [33].

2.3 SDN Security Concerns

As with any new technology, inherent vulnerabilities exist. For SDN, the controller and the OpenFlow protocol are two new targets introduced into the traditional network environment. This leads attackers to target the controller mainly through spoofing, flooding, and modifying. Scott-Hayward *et al.* present one of the most comprehensive reviews of SDN security issues across the application, control, and data layers [34]. With SDN, the controller applies rules and takes action depending on flow tables and logic, while the attacker aims to disrupt the control or action process by manipulating the controller. Controllers are most commonly targeted using unauthorized access, data-leakage, data-modification and denial of service attacks [34].

In [35], Kloti *et al.* apply the Microsoft STRIDE framework to assess data leakage and denial of service impact on SDN. Using Mininet and Open vSwitch for a simulated environment and *scapy* [36] for traffic generation, the authors established flow rules can be discovered by comparing response time of the TCP setup for adding an OpenFlow flow rule. The attacker can potentially learn about network services and hosts without performing a scan. While this attack is mitigated by enabling TLS, OpenFlow does not require this feature.

2.3.1 Applications.

When network control plane programmability was presented to the developer, services previously implemented on middle-boxes were integrated with the controller as applications (or modules). With an unified view and control of network flows, real time configuration allows for unprecedented access to control network traffic. This network-wide view enables algorithms designed to detect, predict, or block malicious traffic at the first sign of untrusted

connections. Although possibilities are bounded only by the imagination of programmers, there are limits to what can be accomplished without impacting network services.

2.4 Advanced Message Queuing Protocol

Since devices are not inherently compatible, events with enterprise-wide implications require coordination. Moreover the optimal messaging system must be OS independent, reliable and secure. With advancement in cloud computing and financial advantage of a distributed architecture, a message-oriented middleware is a viable approach. In 2003, O'Hara introduced an open source messaging protocol with the goal of interconnecting any business system using a broker [37]. Each client can register and communicate using a queue managed by the broker. Following open source development, ISO/IEC 19464-AMQP was released in 2007 [37]. AMQP is a wire-level protocol similar to hypertext transfer protocol (HTTP). Unlike HTTP, AMQP is a binary protocol (intended to be read by machines and not people) and provides reliable queuing, topic based publish-and-subscribe messaging, flexible routing, transactions, and security [38]. These features make AMQP ideal for large scale clusters that require messaging for coordination. Many organizations have adopted AMQP [39]:

- JPMorgan processes 1 billion AMQP messages per day in critical systems.
- NASA uses AMQP for control plane on the Nebula Cloud Computing project.
- Google project Rocksteady uses AMQP to analyze user defined metrics.

AMQP adheres to industry standards for data framing, client-server negotiation and connection handling. Binary protocols are more efficient as they allow more to be sent in each frame verses sending text across the wire [40]. At the application layer, basic command classes exist for session management such as `Basic.Publish` for sending, `Basic.Get` for consuming and `Basic.Ack` for acknowledgment of a message [39]. One of the most significant features of AMQP is the community-driven libraries available in Java, C++, Python, Ada, and Ruby. Figure 7 illustrates a multi-language OS-independent topology of messaging

with AMQP where each agent can communicate through the broker within the server. With such comprehensive coverage, any modern system with support from at least one of the aforementioned languages can use AMQP. As a result of the powerful features provided by AMQP, developers press for native support across all modern OS distributions.

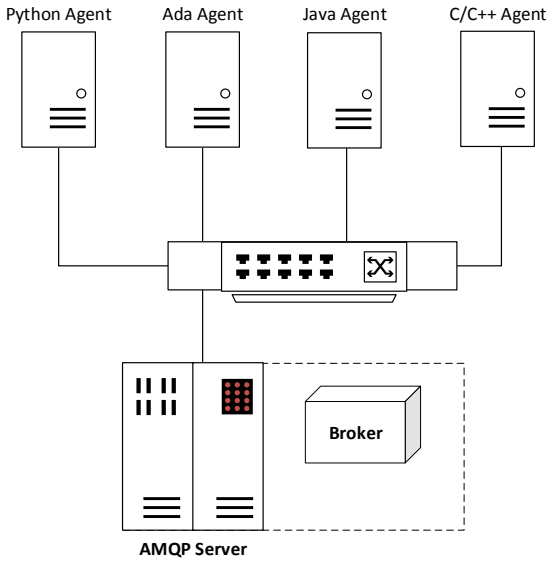


Figure 7. Advanced Messaging Queuing Protocol Language Support

2.4.1 Brokers: Exchanges and Queues.

AMQP uses a modular approach to separate message delivery and storage. Exchanges are the first container used to deliver messages to queues. Each exchange is named and consists of a set of rules to determine where to deliver each inbound message, which is accomplished using a routing key. Messages are stored in the keyed container awaiting a consumer. Queues are the storage location (memory or disk) where messages are held during transmission from one host to another. AMQP uses a series of queues, at least four per queue instance. Each message is in at least one of the four internal queues depending on the state of the message. For example a message might be in memory or on disk, each stored as a unique internal sub queue.

Figure 8 illustrates the topic exchange configuration used in this research. P1-P5 repre-

sent the hosts (producers), each with an agent sending messages (events) to the broker. At the exchange, depending on the routing key, messages are placed in the appropriate queue– in this case *SecurityState*. C1 is the consumer (module within the controller) retrieving messages from the SecurityState queue.

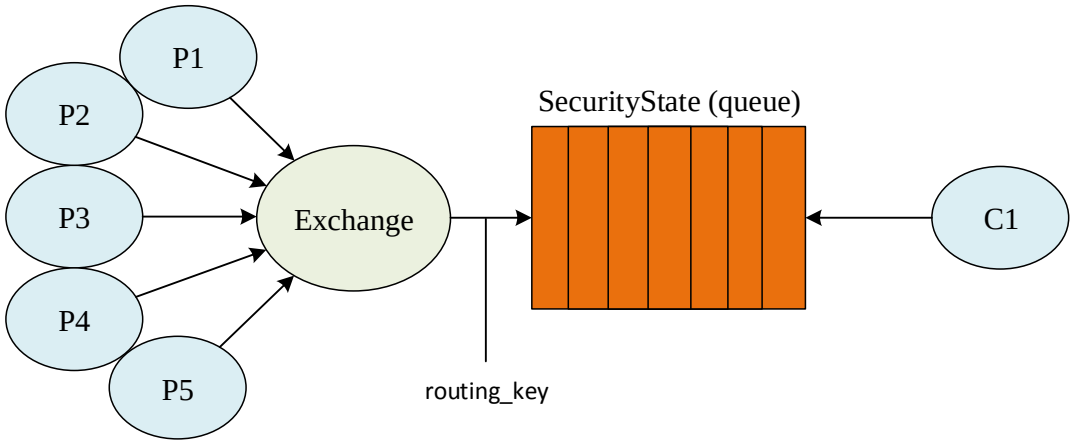


Figure 8. AMQP Topic Broker Example

2.4.2 RabbitMQ.

RabbitMQ is a highly customizable and widely adopted implementation of AMQP. Key features include node-clustering, highly available queues, multi-protocol support, message tracing/tracking and a plug-in system [41]. RabbitMQ supplies a management UI for monitoring and control of the message broker. AMQP libraries for most languages are freely available through open source projects; the one selected for this research is the Python library *pika* [42]. A test scenario with a single-host three node cluster (three Erlang nodes running RabbitMQ) with 10 publishers and 10 consumers (both remote from broker) resulted a publish rate of 33.3K messages per second and a consume rate of 16.2K msg./sec. [43]. RabbitMQ is used for this research because it is open source, widely adopted by industry leaders across multiple disciplines (Google, NASA, JP Morgan), provides secure messaging, logging, and tracking, and is capable of millisecond end-to-end message delivery.

2.5 Dynamic Security Control Using SDN

Dynamic Security Control Using SDN (DSCS) has five components (Figure 9):

1. Host agent detection and updates (Section 2.1.4)
2. Reliable host to SDN controller messaging using AMQP (Section 2.3)
3. SDN controller with DSCS module (Section 2.2.5)
4. Role-based flow profile (Section 2.1.5)
5. Flow table updates from controllers to switch (Section 2.2.2)

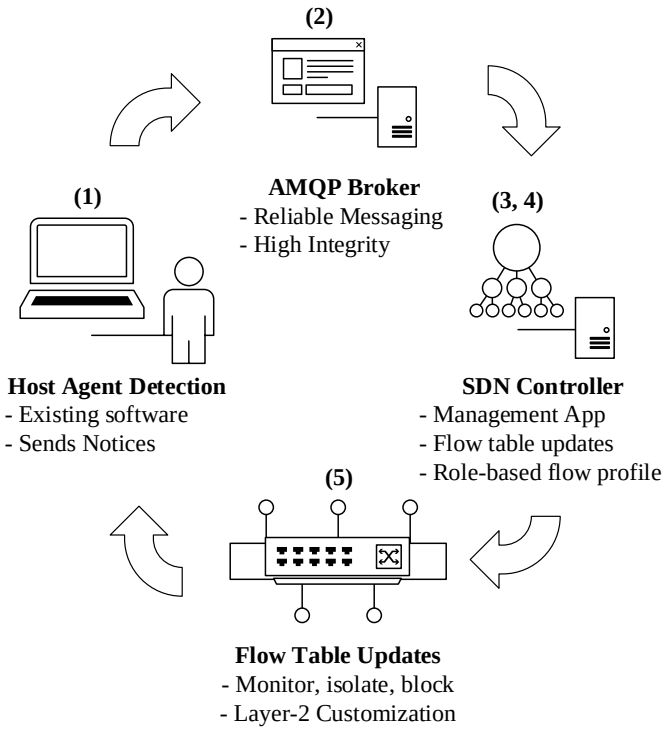


Figure 9. Dynamic Security Control Using SDN Process

Dynamic or automated network security strives to preemptively adjust policies to prevent the unknown. That is, the next zero-day or exploit without a signature. While advanced antivirus software is capable of identifying and taking action on hosts, little is done to adjust the logical topography (flow tables) of the network. Assuming each host is unaware of the security status of other hosts, the rest of the hosts within the enterprise are still vulnerable to the attack and all communication to and from the compromised host are allowed to

continue. Figure 9 shows how each host provides security event data to the controller and updates are applied to the network. It is important to note this model is not specific to any single product but a process employed using the aforementioned types of technology. Floodlight is selected for this research because it is a popular and community-supported controller platform. However, DSCS can be implemented with any platform capable of flow table updates, custom applications, and the AMQP library.

2.5.1 Related Research.

One similar proof of concept project, Active Security, implements a network IDS that communicates with the SDN controller to trigger an agent to take action on the host. The host collects an image of memory (RAM) and sends it to a forensics server for processing. If malicious connections are detected, the forensic server communicates with the SDN controller to block those outbound connections. This process is focused on network IDS triggering and blocking malicious connections for the specific host [2].

While this approach is effective it fails to manage the continuum of security events (Section 3.1.3). Low confidence security events might not require memory collection and blocking, but are important to the overall security state of each host. This research proposes a more general approach to handle security level changes that might not warrant out right blocking. From routine to major security incidents, the state of each host is monitored through agent-controller updates. Unlike Active Security, DSCS uses secure messaging (AMQP SSL) between host and controller.

2.5.2 Chapter Summary.

This chapter introduces security principles, software defined networks, and messaging using AMQP to support the implementation of the DSCS framework. Additionally, how SDN has provided a new approach to network management is presented.

III. Framework Design

3.1 Overview

This chapter provides a detailed description of the component parts used to implement the DSCS system. In order to achieve the goals of efficient, effective, and ease of implementation, each component is developed using open source software and tools vetted by industry leaders in networking and security. After considering the technologies previously discussed, the four components for the DSCS system include an agent, role-based flow profiles, a broker, and an SDN controller (Figure 9). The *role-based flow profile* (Section 3.4) is a novel contribution that enables host-level customization of access based on communication requirements and profile (workstation, server, etc). Additionally, the use of a security index representing the trustworthiness of each host is unique. The following sections describe in detail the agent, role-based flow profile, AMQP messaging, and Floodlight controller.

3.2 Design Parameters

The following design parameters are considered while developing the DSCS framework:

Efficiency

1. Modifies network flow at layer-2 in less than 1 second.
2. Only malicious network connections are blocked, not critical services.
3. Automated detection, reporting, and response using flow table updates.

Effectiveness

1. Every security event triggered is processed without fail.
2. Routes to critical hosts are maintained and blocked automatically.
3. Malicious outbound connections are blocked at layer-2.

Ease of Implementation

1. Agents interface with existing security software (e.g., antivirus, HBSS).
2. System (Controller and Broker) do not require high performance computers.
3. Compatible with existing host-based security systems.

3.3 DSCS Design

This section provides a high-level description of how the DSCS system is designed to operate. Agents interface with existing security software like antivirus on the host using a custom Python interface created by the developer (e.g., Symantec's API is Symantec Scan Engine 5.2 [44]). Within the interface, each class of security event provided by the API is defined and assigned a message type and action (e.g., malware: block, registry: block, authentication: notify). Following the example illustrated in Figure 10, *Agent 1* is configured to interface with the Symantec Scan API running on the host and infected with malware. Since malware detection triggers an event to send an update to the controller indicating the host is no longer trusted, a message is sent to the *AMQP Broker RabbitMQ* (orange lines). The controller is continuously checking for new messages (blue lines) and consumes the message upon delivery. The message (event) is processed by the *SDN Controller DSCS Module* and pushes a flow table update to all connected switches to block traffic from *Agent 1* (green shields). The following sections explain each component in detail.

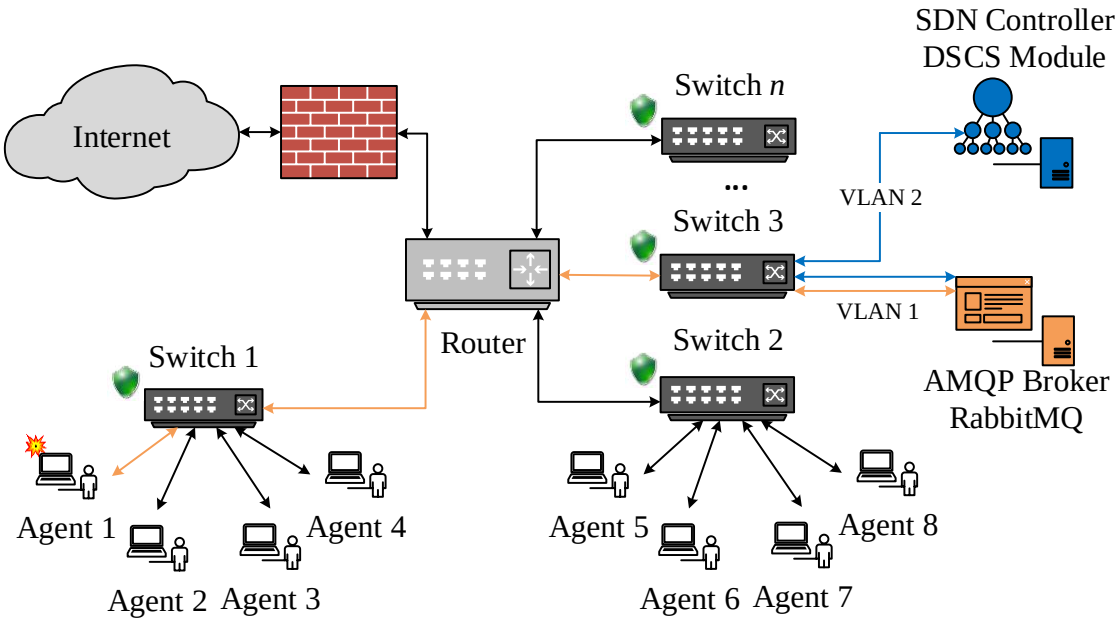


Figure 10. DSCS System Design

3.4 Host Agent

The host agent is implemented as a system of Python scripts (version 2.7.2), Java, virtual machines, and a virtual switch. To test the robustness and scalability of this system, real world security events are simulated. The agent operates under the following assumptions and limitations. First the host is running a host-based security system with an accessible API–this feature is simulated during testing. Second, since the agent sends AMQP messages through the broker, sufficient rights and permissions to query and use network devices are required. Next, the host security system can identify and report malicious connections as an (IP PORT) pair using a loaded module, in this case *ModSysStatus* (Section 3.3.3). Finally, during install the agent is configured by an administrator who specifies values for critical routes, a confidence value for each alert type (malware, registry, authentication, etc.), and a profile type (Section 3.4.2).

Figure 11 models the states within the host agent program. The initial state S_0 identifies the start of the code. After setup, S_1 is the core of the code where events are detected and a message is constructed and sent to the controller via the broker. Appendix A provides a full inter-component state digram.

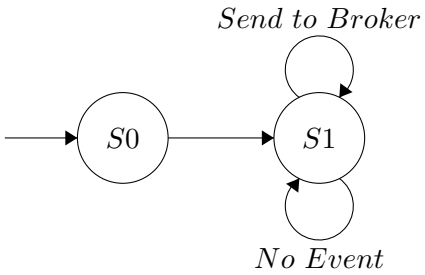


Figure 11. Agent State Model

The agent is comprised of a main subroutine and set of loaded modules. In Figure 12, each module provides custom functionality based on the purpose of the application, in this case managing the control plane of the network using ModSysStatus.py. Another example is a module to perform incident response actions upon security status change, e.g., capture

memory or other volatile data. The modular design of this agent allows for it to attach to any existing host-based security system with an accessible API.

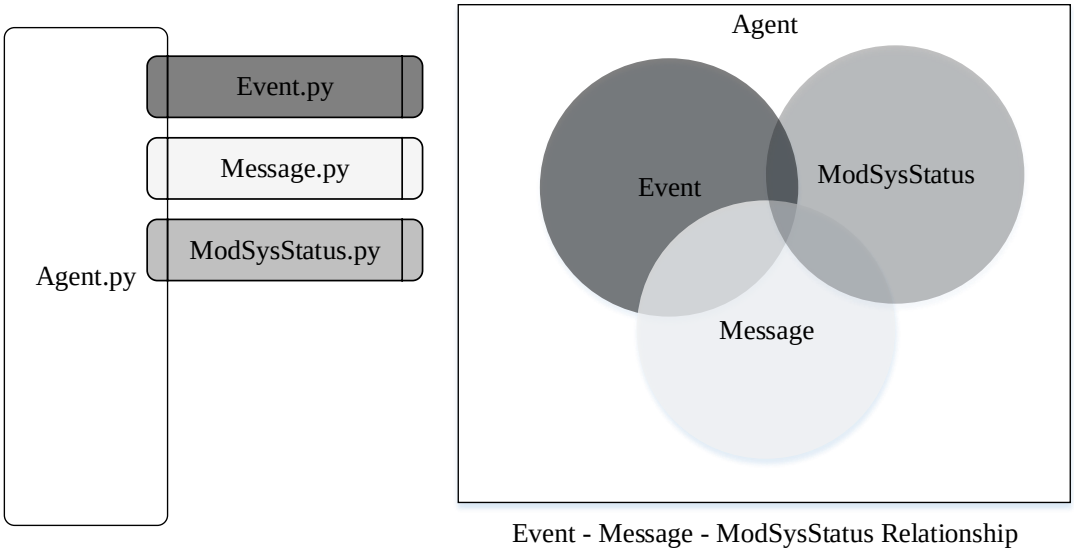


Figure 12. Agent Design Model

3.4.1 Security Event Detection.

The simulated security event types built into this system are provided through coordination with 24 Air Force and reflect four top event classes detected on the AFNet. These event types exist in all environments but may vary based on organizational requirements. However, each event does span any infrastructure running Windows based operating systems. As discussed in Section 2.1.2, an event is comprised of an *action* and a *target*. During testing, events are generated with a confidence setting from zero to one based on the administrators trust in the signature. The confidence setting is used to determine if a blocking rule should be implemented. The SDN module is configured to take action based on the event type and confidence setting. This model allows for tracking the overall security state at a host based on events or event clusters (e.g., series or groups of events). Section 3.6.2.1 provides detail on how the SDN module manages this process.

3.4.1.1 Event Generation.

Events are generated by the test suite and consist of four state variables. The first is a binary decision if the event will fire or not (*fireEvt* 0 or 1). The second consists of four levels (*evtTyp* 0 - 3) representing the events described in Table 1 as `(0:Malware, 1:Unauthorized Media, 2:Authentication, 3:Registry)`. Next, to simulate real-world delay in events, a delay of zero to two seconds (*evtDly* 0.00 - 2.00) is applied at each iteration per host. However, since the purpose of this research is to test how the system responds under a greater number of events per second, a delay of zero is used for all tests. Finally, a confidence value originally set by the network administrator is attached to the event (*evtCon* 0.00 - 0.20). Each event is assigned a confidence value to represent the network administrators trust in the detection method triggering the event. If the event warrants a block to occur, the confidence is set 1.00 (e.g., malware detected). While events with lower severity might only receive 0.10 (e.g., unauthorized media access). Since events are randomly generated only those not causing a block event use *evtCon*. For example, if the detection system is highly targeted to a specific malware activity the confidence setting might be `1`, but if the event is only a small indicator of *possible* malicious traffic, the setting might be 0.01. Even the smallest event type is significant when considering the overall security state for a host. If the event does trigger, the event type flag is set, a confidence value is generated and a message is sent to the AMQP broker where it is stored until being consumed by the controller.

3.4.2 Message Data Structure (Message.py).

This module facilitates communication with the SDN controller. The Message class produces a Javascript String Object Notation (JSON) object with information required by the controller for processing events. Fields for the message structure include `msgtype`, `hostID`, `MAC`, `IP`, `msgData`, and `updates[]` (an array of objects). The message object is designed to provide information to block or allow flows, though any structure could be passed to the controller for processing. In Figure 13, `msgData` contains the command being

Table 1. Security Events

Event	Action/Target	Flow table update
Malware: Virus scanner or other signature based device identifies malware on the host.	**Action:** Bypass, Modify, Authenticate **Target:** Account, Process, Component, Computer, Network	Yes (critical hosts only & notify security officer)
Unauthorized Media: User attaches external media or burns a CD/DVD	**Action:** Bypass, Read, Copy, Steal **Target:** Data	Yes (Block out bound & notify security officer)
Authentication: Excessive failed logins, account locked out, unexpected login method (remote, console)	**Action:** Authentiate, Bypass **Target:** Account	Yes (notify security officer & revoke account)
Registry: Targeted/critical areas of windows registry is modified.	**Action:** Modify, Authenticate, Bypass **Target:** Account, Process, Component, Computer, Network	No (notify security officer)

sent "modify" and the `updates[]` object contains two rules to apply–allow 10.0.0.3 and 10.0.0.4 to communicate. Upon consuming this message (event), the controller takes action if the host is trusted and allowed to make this modification (based on flow profile assigned).

3.4.2.1 AMQP libary pika.

To facilitate AMQP communication to the agent, pika [42] is used to manage the connection with the AMQP broker. The components used by pika include credentials, connection parameters, channel management, and sending/receiving messages (basic_publish). Credentials to authenticate with the broker depend on a username and password. The encrypted credentials are passed during connection negotiation but are stored in plaintext in the agent configuration files. For this implementation, the username is client$N$, where N represents a number from 0 - 9, and the password is 12 characters randomly generated using the Linux program APG [45]. The connection parameters are specified within the agent to specify host, port, virtual host, credentials, and SSL (shown below).

```
self.parameters = pika.ConnectionParameters('10.1.0.179', 5671,
        '/', pika.PlainCredentials('client1', 'EJ1j0IrIy3b$'), True)
```

```
{
   "msgType":"event",
   "hostID":"836b389deecf32f1cbee169aa6e3967b",
   "MAC":"2c:44:fd:66:2b:c2",
   "IP":"10.1.0.59",
   "msgData":{
      "command":"modify"
   },
   "updates":[
      {
         "src-ip":"10.0.0.3",
         "dst-ip":"10.0.0.4",
         "priority":10,
         "action":"ALLOW",
         "conf":1
      },
      {
         "src-ip":"10.0.0.4",
         "dst-ip":"10.0.0.3",
         "priority":20,
         "action":"ALLOW",
         "conf":1
      }
   ]
}
```


Figure 13. DSCS JSON Message Structure

Connection parameters may also be presented as a URLParameter in the form: `amqp://client:client@localhost:5672/%2F`, known as the AMQP URL.

As an event fires, a secure connection (SSL) is established with the broker using the aforementioned credentials, connection parameters, and channel options specified within the message module. After a connection is established, the channel object declares the exchange, routing key, and message body. Delivery confirmation is set to ensure all security related messages are not lost in transit and reach the broker.

3.4.3 System Status (ModSysStatus.py).

This module runs on the host and acts as an extension to the SDN application running on the controller. In a production environment, this module is multi-threaded and interfaces with host-based security software to generate system security status updates to the controller. As detection is not the purpose of this research, ModSysStatus uses the DSCSTest module to simulate security events.

3.4.4 DSCS Test Suite (DSCSTest.py).

This module is the primary portion of logic for the agent during testing, responsible for executing tests and collecting performance data for the host. As described in Section 3.3.1.1, *fireEvt*, *evtType*, *evtDly*, and *evtCon* are randomly generated to account for event variability expected to occur during a real world implementation. When initialized within the ModSysStatus module, DSCSTest requires the agent (object Agent.py), a sample size, and number of runs as parameters. During testing the doRuns() method is executed where the firewall is enabled and the host is registered with the controller as a node in the network. Each host is uniquely identified by generating an MD5 hash of the hostname concatenated with the MAC address. Table 2 shows the structure for message commands used by the agent.

3.5 Role-Based Flow Profile

Using the principle of RBAC, where each subject is assigned a role, hosts register with the controller using *registerAgent()* and specify a profile type. The general profiles supported are *server* and *workstation*. Profiles are unique based on initial rules installed, static routes maintained, and how the node is treated during a critical security event. Applying roles to a network and flow enforcement policy allows for the controller to maintain state information about each host registered. This implementation of DSCS tracks security status of each host by maintaining a security index for each host. The index is updated as *sendSecUpdate()* is executed by the agent. This index represents the security risk associated to the host–higher

Table 2. Message Structure

Function	Structure
initFirewall()	{"msgType": "setup", "hostID": "xyz", "MAC": "2c:44:fd:66:2b:c2", "IP": "10.1.0.59", "msgData": {"command": "toggle"}}
registerAgent()	{"msgType": "setup", "hostID": "xyz", "MAC": "2c:44:fd:66:2b:c2", "IP": "10.1.0.59", "profile":"workstation"}
sendPollUpdate()	{"msgType": "polling", "hostID": "xyz", "MAC": "2c:44:fd:66:2b:c2", "IP":"10.1.0.59"}
sendUpdate()	{"msgType": "event", "hostID": "xyz", "MAC": "2c:44:fd:66:2b:c2", "IP": "10.1.0.59", "msgData": {"command": "modify"}, "updates": [{"src-ip": "10.1.0.59", "dst-ip": "0.0.0.0", "priority":10, "action":"DENY", "conf":1}]}
sendSecUpdate()	{"msgType": "event", "hostID": "xyz", "MAC": "2c:44:fd:66:2b:c2", "IP": "10.1.0.59", "msgData": {"command": "sec-update"}, "updates": [{"conf":0.182324745426}]}
sendDelete()	{"msgType": "event", "hostID": "xyz", "MAC": "2c:44:fd:66:2b:c2", "IP": "10.1.0.59", "msgData": {"command": "delete", "arg0":"123"}, "updates": []}
purgeData()	{"msgType": "purgeData", "hostID": "xyz"}
sendSyncData()	{"msgType": "syncData", "hostID": "xyz", "data": [{"id":"0.999927520752"},{"val":"1.00016593933"}, ... , {"val":"0.999927520752"}]}

index implies higher risk.

3.5.1 Managing False Positives.

Dealing with false positive alerts is a significant challenge for automated security systems. To avoid denial of service to hosts with many low confidence events, blocking only occurs when confidence reaches a certain threshold determined by the network administrator during installation. This occurs as the DSCS module within the controller maintains a security index for each node registered. While many low confidence events may raise the node security index above an acceptable limit, the node will still maintain access to resources unless administrative action is taken. This approach highlights hosts that are a potential risk based on the activity reported by the agent before blocking all communications.

3.5.1.1 Server.

The *server* profile is focused on being available even if the host is compromised. This is achieved by identifying critical routes and white-listing core systems by IP:PORT combination or MAC address. Servers are not expected to establish outbound connections to unknown hosts or connect to hosts within the same network without white-listing. The most unique feature for the *server* role is based on providing services even if the underlying server is compromised. This is accomplished by not blocking inbound connection requests on published services unless a critical stop is triggered. In the interest of not denying service based on false positives, this system allows a compromised system to provide critical services such as authentication or electronic mail.

3.5.1.2 Workstation.

Workstations are expected to be able communicate with anywhere on the public Internet. This limits outbound blocking on common ports such as 80 or 443. This is a key reason why attackers use reverse connection techniques to avoid detection during an attack. Unlike the server profile, if a high confidence modification with a DENY update is triggered the workstation will be blocked from accessing the remote address. If the event is critical, access to external and non-server hosts are blocked as well.

3.6 AMQP Messaging

The second component in the DSCS process is secure reliable messaging. Using AMQP with SSL enabled through the pika library (Section 3.3.2.1) each node within the network has a secure channel to pass messages to the SDN controller through a RabbitMQ message broker. On the broker, an exchange containing message queues facilitates the delivery of messages from each agent to the SDN application. This method is reliable because AMQP facilitates delivery confirmation and tracking of each message to ensure fidelity. DSCS employs a publisher-consumer model where many nodes publish to the same queue with a common routing key and a single consumer, the SDN controller, processes the messages as

they become available. Figure 14 illustrates the states within the AMQP broker RabbitMQ. S_0 represents the initialization of the program. As a message is sent or received, S_2 and S_3 validate the message and return to the main loop S_1. Appendix A provides a full inter-component state digram.

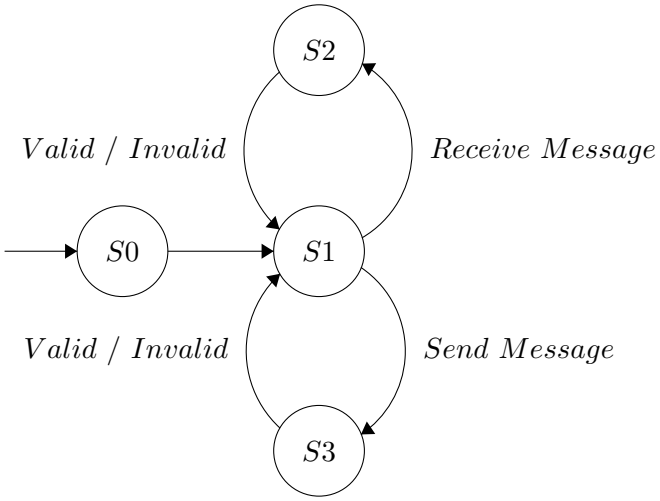


Figure 14. Broker State Model

3.6.1 Broker, Channel, Exchange, Queue.

The broker functions similarly to a mail collector responsible for managing *exchanges* (mail room) and *queues* (mail boxes) organized within containers called virtual hosts (post offices). The default virtual host is '/', and can be considered a set of databases (exchanges or mail rooms). The broker used in this implementation is RabbitMQ 3.5.4 Erlang 18.0 [41]. A secure TCP channel is negotiated between each host and the broker. The two channel commands used in this implementation are *confirm_delivery()* and *basic_publish()*. The first used to verify message receipt and the second to publish a message to the broker specified in the channels connection parameters. Before a message is sent to the broker, it must be addressed to a recipient. Using a publisher-consumer model, the parameters are set on the client in the *pika.ConnectionParameters* and *channel.basic_publish* objects.

The type of exchange used for the DSCS system is a *topic exchange*, which is optimal for delivery to one or many queues. Queues are the resting place for messages in transit and are persistent in the broker. This means any message that arrives in a durable queue will not be lost if the broker crashes or the service is restarted. For this reason, the DSCS queue is configured as durable and non exclusive-available to many connections and does not delete after disconnect.

3.7 Floodlight Controller

While there are many SDN controllers with modular application development support, Floodlight is used because it is a Java-based implementation with open source libraries to AMQP, RabbitMQ, network socket programming, and RESTful application support. Floodlight version 1.1 is extended to develop a network security service to track the security status, using a *security index* (configured within the Network.java class as SEC_IND_LIMIT), of the nodes connected to an SDN managed switch. The service provided by DSCS is implemented as a module running on top of the core SDN controller and uses the Stateless Firewall developed by Tahir, Wang, and Izard of Big Switch Networks [46]. Figure 15 provides a general Unified Modeling Language (UML) representation of the classes created to extend the Stateless Firewall.

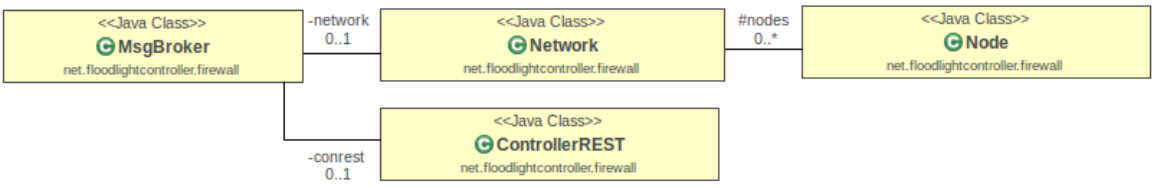


Figure 15. DSCS Java Implementation UML

3.7.1 Developed Components.

The Floodlight Firewall application is modified to initialize a multi-threaded *MsgBroker.class*. MsgBroker enables the firewall to accomplish three new services. First a represen-

tation of the network is maintained in a network object (*Network.class*), containing nodes and links. This object is maintained by the MsgBroker and provides real-time security status for all nodes registered. Next, a channel to the message broker is established for consuming messages from agents located on each host. This channel is implemented using the *com.rabbitmq.client* library [47]. If messages are pending delivery on startup, they are processed after the firewall module is initiated and not lost. Finally, the *ControllerREST.class* allows for modification of firewall rules using the existing REST interface provided by the Firewall module. With these three services, the existing Firewall.class is able to maintain a security state on all nodes in the network.

Figure 16 models the states within the Floodlight controller application DSCS. As messages are consumed from the broker, the MsgBroker class updates the network map (Network class) and individual node security status (Node class) while also managing the channel between controller and broker (RabbitMQ library). S_0 represents the start of the DSCS application followed immediately by entering state S_1. S_2 signifies the use of the ControllerREST class to update flow tables on connected switches. S_1 is entered after processing each event. Appendix A presents a full inter-component state digram.

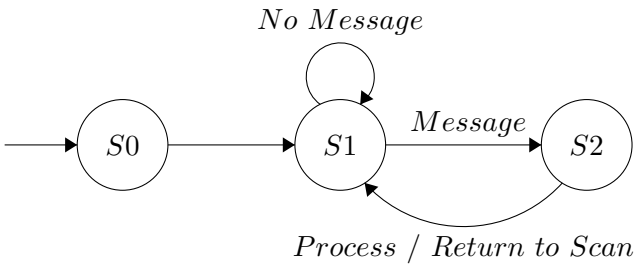


Figure 16. Controller DSCS State Model

Msg.class. As MsgBroker consumes a message from the broker, the payload is parsed using the Msg class. *MsgType*, *hostID*, *MAC*, and *IP* hold the identifying data used when processing the message after parsing. Two hash maps, *msgData* and *updates*, hold the event details to update the network and if required, firewall rules. The *msgData* map contains commands and any arguments required e.g., *command: modify, sec-update,*

toggle. Command-type *modify* indicates the *updates* object has entries to be considered for application to the firewall. While *sec-update* indicates an event to modify the security index occurred. Finally, the *toggle* command is used to enable the firewall during setup.

3.7.1.1 MsgBroker.class.

As shown in Figure 17, the MsgBroker class contains components to maintain a network map, interface with the AMQP broker, and execute changes to the SDN controller using the RESTful Firewall interface.The network (Network) object represents a map of nodes registered through an agent on the network. All registered nodes are in the same network and data is not shared between controllers in this implementation. ControllerREST enables communication with the Firewall REST application and is described in the next section. The static final variables (QUEUE_NAME, ROUTING_KEY, EXCHANGE, EXCHANGE_TYPE, DURABLE, USER, PASS, HOSTNAME) are for the configuration of the AMQP broker. The *checkNodes* variable is a boolean value indicating if nodes should be checked periodically (default TRUE).

3.7.1.2 ControllerREST.class.

Table 3 presents the commands used in this implementation [48]. Figure 17 illustrates the structure of the ControllerREST class. Depending on the method called, the POST, PUT, GET, or DELETE strings are used to issue a REST command to the Stateless Firewall module. The data string holds the command being sent to the controller when a POST or GET method is executed.

In order to update the firewall rules applied as flow table rules on each switch, this class provides methods to execute REST commands. Since the Floodlight Stateless Firewall is a RESTful application, this class acts as an interface to mange the underlying storage source table containing firewall rules (*net.floodlightcontroller.storage*). As the source table is updated, modifications are pushed to all connected switches. Depending on the command and action sent from each host agent, the *sendPost()*, *sendPut()*, or *sendDelete()* commands

Table 3. Floodlight Stateless Firewall RESTful API

Uniform Resource Identifier (URI)	Method	Description
/wm/firewall/module/status/json	GET	Firewall Status (enabled/disabled)
/wm/firewall/module/enable/json	PUT	Enable firewall
/wm/firewall/rules/json	PUT	Disable Firewall
/wm/firewall/rules/json	GET	List all rules installed
/wm/firewall/rules/json	POST	Create new rule

Sent using HttpURLConnection (java.net.HttpURLConnection) with URI and method.

are executed to manage the Firewall configuration. The data string is the payload sent during a POST or PUT method. For example, as a *modify* command is issued, data is set to a JSON string object like seen in *sendUpdate()* in Table 2 and a POST command is sent to the URI `/wm/firewall/rules/json`.

3.7.1.3 Network.class.

The Network class maintains a real-time representation of the network by documenting the unique hosts registered and their associated security index. Upon registering, a Node object is created and added to the *nodes* list. During the *checkSecurity()* method, the security index for every node is compared to SEC_IND_LIMIT which is set by the network administrator. If the index value exceeds SEC_IND_LIMIT the host is flagged as no longer trusted. In a real-world implementation, since the security index increases as events occur on each host, an acceptable index limit must be determined based on the hosts unique configuration. Figure 17 provides a UML description of objects and functions of the Network class.

Depending on the profile set during agent configuration (e.g., workstation, server), the staticLinks structure holds a hash map of IP addresses keyed by the hostID. This structure allows for a representation of the network configuration for reference when making changes to the firewall. If the IP is added to this list, it should not be blocked irrespective of the hosts security status. This feature allows mission critical resources to be available even during security events that otherwise would stop all communications.

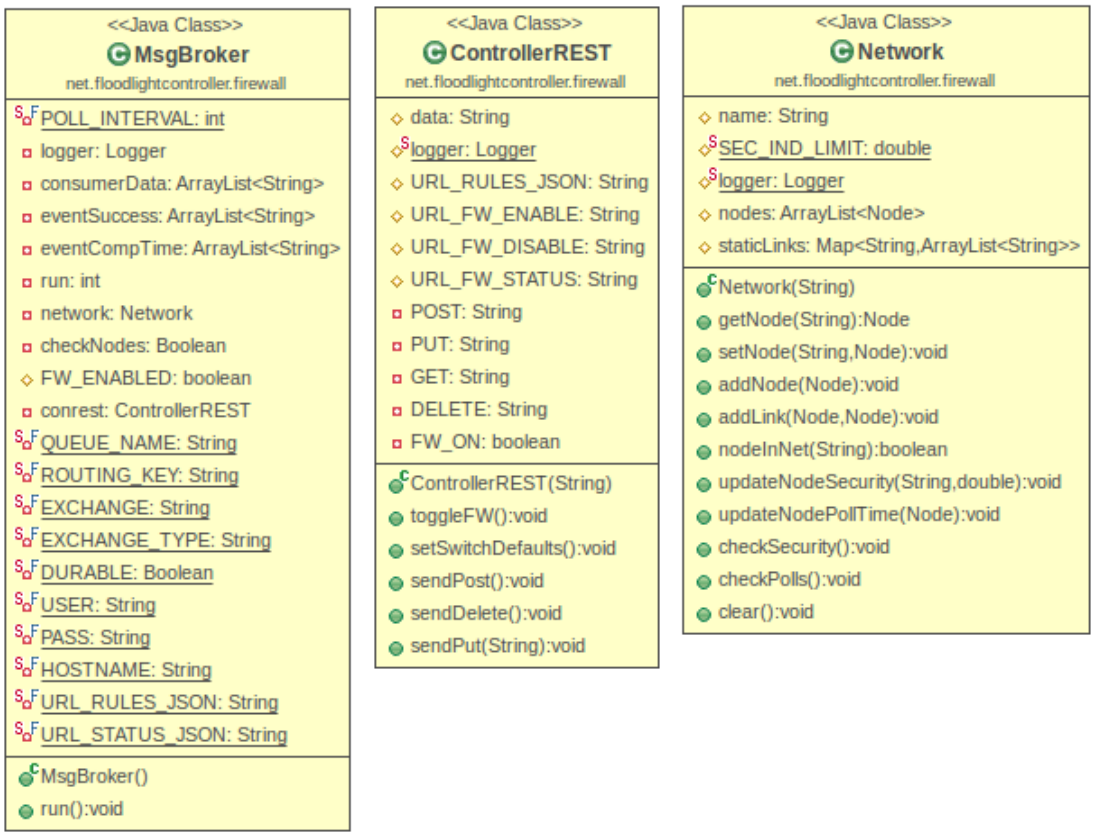


Figure 17. UML Definition of Developed DSCS Module

Node.class. The Node class represents a registered host in the network. A node consists of a hostID, IP address, MAC address, security index, a list of allows and blocks, and *pollTime* (the last time the node checked in). Four objects are used to keep track of time for each specific event. *MsgDeliverAgentLog* records the elapsed time to deliver the event message from agent to broker. *ConsumerDataLog* records the elapsed time to consume a message event from the message broker. *EventCompleteTimeLog* records the time required to send an update to the firewall and then be applied to the switch. Finally, *EventSuccessLog* records if the action sent to the controller was completed or not. These four pieces of data are used to determine the metrics for the DSCS system during each trial.

3.8 Design Summary

In summary this chapter describes the DSCS process and design. The framework is designed to be efficient, effective, and easy to implement. This design is a novel approach at automating firewall rules applied at layer-2 using a Floodlight stateless firewall, an AMQP message broker, and security events sent by an agent on individual nodes.

The features unique to DSCS include:

- dynamic updates to layer-2 flow tables
- node security index tracking at controller
- role-based flow profiles to ensure mission critical hosts are reachable

Initial pilot tests produced a mean of 24 milliseconds from the event triggering to flow table update. The test environment included two hosts, one AMQP broker, one SDN controller, and one virtual switch.

IV. Methodology

4.1 Problem Statement and Goals

This research focuses on a process to rapidly respond to host-level security events by using SDN flow table updates, role-based flow classes, and AMQP messaging. Aimed at assessing if effective dynamic layer-2 management is possible, role-based flow profiles and existing security software modifies layer-2 flow rules using an SDN controller application.

The goals in testing this implementation stem from those stated in Section 3.2. For each, the following questions are important:

Efficiency.

- Can a network flow be blocked automatically in less than 1 second of a security event firing from a host?
- How does the DSCS system respond as the number of events increase? (e.g., mean time for flow table updates)

Effectiveness.

- Is every event sent to the broker processed?
- Are routes to critical infrastructure maintained after flow tables updates are made?
- Are firewall rules to block or allow traffic effective at layer-2?

Ease of Implementation.

- Is the process OS independent?
- Does the system require more configuration than existing security solutions?

4.2 Approach

The DSCS framework notifies network control devices of security events occurring on hosts. Events are sent using an AMQP message broker and consumed by an SDN controller application where flow table updates are issued using OpenFlow. See Chapter 3 for a detailed explanation of DSCS.

To test this framework, an environment with hosts, switches, brokers, and controllers is required. Starting at the host, an agent is developed to simulate security events based on the most frequently occurring on the AF network. These events include malware detection, unauthorized meida, authentication, and registry modification. To simulate these events, upon initialization the agent uses a list of randomly generated security events provided by the DSCSTest.py module and sends an encrypted message to the SDN application through the AMQP broker. The SDN application retrieves queued messages and determines what flow table updates are necessary. Depending on the scenario and the host's flow profile, OpenFlow updates are sent to all managed switching devices. Agents can only request flow table updates to the IP address of their assigned host, and the controller will only implement updates from trusted hosts.

Measurements are taken to quantify the Message Delivery Time (MDT) (from host to SDN application), Event Completion Time (ECT) (time from event triggered to flow table update), and Event Success Rate (ESR) (if the changes accomplished the desired effect). These metrics are analyzed to determine how fast and effective flow table updates are made based on dynamic host events. The number of hosts is held static at 5 during testing due to system resource constraints, however the number of events is manipulated from 200 to 10000 per host. Since the application at the SDN controller could theoretically be implemented at any SDN controller, Floodlight Section 2.2.6 is used since it is well established and has a firewall module available.

4.3 System Boundaries

As shown in Figure 18, the system under test (SUT) is the DSCS process. The components of the system are the host agent (Agent.py), AMQP broker (RabbitMQ non-clustered), Floodlight SDN controller, and the DSCS application. These components under test (CUT) produce metrics for evaluating the system. The workload into the system is a set of randomized events.

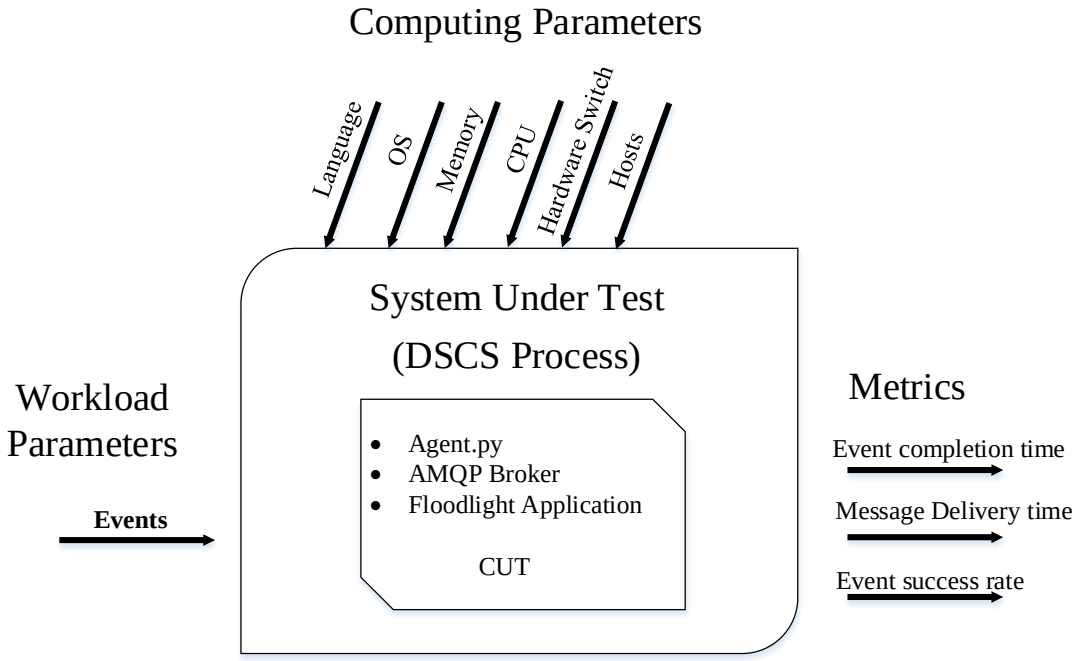


Figure 18. System Under Test

4.4 Parameters and Factors

This approach uses workload and system parameters. Parameters that vary during the experiments are called *factors*. The range of values each factor can hold are called *levels*. This section defines and explains each parameter and how it relates to the SUT. The parameters held constant include the programming language, OS, memory, CPU, hardware switch, and number of hosts. Table 4 presents factor levels for the number of events per

host used in both experiments.

Table 4. Factors

Control Variable	Experiment	Tested Range	Tested Levels
Events	1	1000-50,000	1,000, 5,000, 10,000, 50,000
Events	2	10,000-50,000	10,000, 50,000

4.4.1 Workload Parameters.

4.4.1.1 Events.

This parameter represents the number of events occurring in the network. These events are created from each host running the agent and communicate through the message broker to the SDN controller. Events are generated randomly and contain an encrypted text string of approximately 240 bytes with the pertinent event data. The number of events is specified in the agent configuration and how they are generated is explained in Section 3.3.1.1.

4.4.2 System Parameters.

Figure 18 identifies the computing parameters as language, OS, memory, CPU, hardware switch, and hosts. The agent is written in Python, however, any programming language with the AMQP library is sufficient. Since the DSCS process is OS independent, Windows 7 is used to represent the most common host on a production network. Next, memory capacity and CPUs for each host is set to 4 gigabytes and 2 cores, respectively, allowing for minimal delay due to system resources. All traffic routes through a hardware switch (HP5900) and is manged by the Floodlight controller DSCS module. Finally, 5 hosts are used to send different levels of events.

4.4.3 Performance Metrics.

This research focuses on three performance metrics. ECT measures how quickly the system can react to a security event detected on a host. More specifically, the wall clock

time from when the event is triggered to the flow table update on the switch. As in a real world implementation, not every event will require a flow table update. However, all events are included when calculating metrics.

MDT measures the time taken for messages transfered from host to controller through an AMQP broker. This metric presents a view on how fast and resilient secure AMQP messaging responds under different levels of events. The AMQP protocol is robust and is not expected to fail even under high load.

ESR measures the rate messages are triggered, transmitted, and successfully executed. Each message (event) is tracked from host to SDN application. If the event requires a flow table update, a check is performed to verify the appropriate change was applied. Upon verification the check is a PASS, otherwise it is a FAIL. The ratio of the sum of PASS to total events triggered is the final event success rate.

4.5 Experimental Design

4.5.1 Overview.

Two experiments are designed to address the goals posed in Section 4.1. The SUT is evaluated for efficiency and effectiveness by testing different numbers of events. To minimize experimental bias, identical systems are used to host all virtual machines and each host has a unique network interface card (NIC). Additionally, all hosts are identical within their respective OS version and have the same alloted CPU and memory values. Each event is randomly selected to ensure all event types are considered equally and not selected with bias. Tests are automated using the scripts in Section 4.7.2 and the controller is restarted between each test to ensure system state is the same during each run.

4.5.2 Data Processing.

Figure 19 provides a high level view of the process used to evaluate data captured during each experiment. For each run, all data are consolidated into a single data file (e.g., 1000_r1.csv contains the first trial of 1000 observations). After all trials are complete, the

data are parsed using R into one table per file. The mean for each metric is calculated for each table and stored in a single table with 30 entires. This final table with the mean of means is represented using tables, plots, and if appropriate a linear model. This approach of using sample means is applied because the data collected is not normally distributed. The Central Limit Theorem asserts that given a sufficiently large sample size, samples means are approximately normally distributed [49].

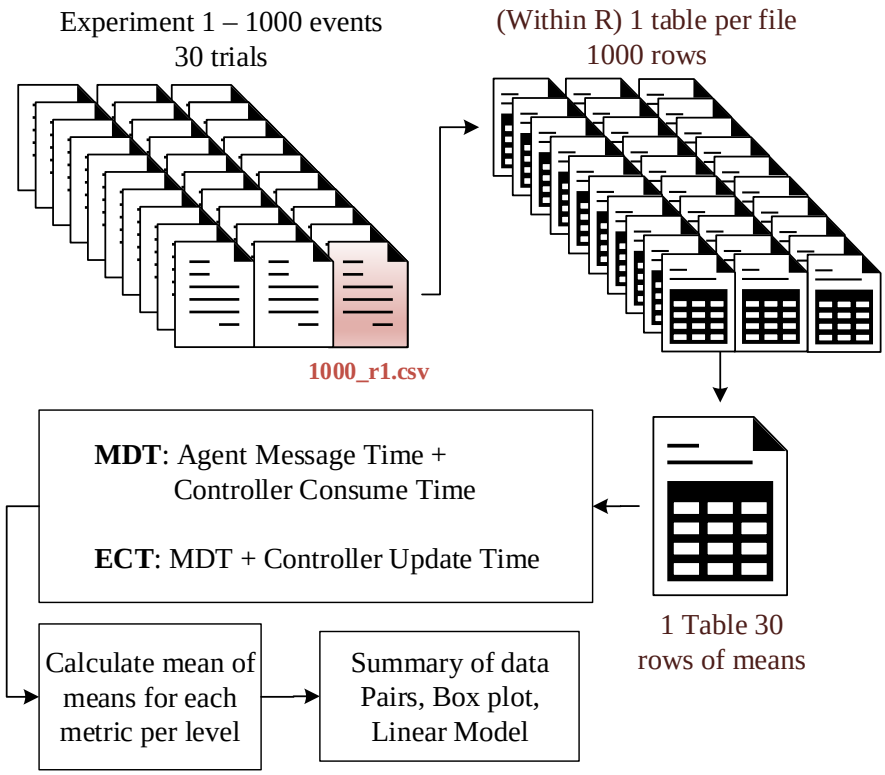


Figure 19. Data Processing Diagram

4.5.3 Determining Sample Size.

During experimentation a two sample power t-test using R [50] is used to determine the number of samples to achieve a 95% confidence of one millisecond difference with 90% power. With the observed maximum standard deviation of 3.65, approximately 281 events per group are required to achieve 95% confidence (R Code in Appendix B). Given this

information, both experiments use event levels starting at 1,000 and increase up to 50,000.

4.5.4 Test Timing.

Synchronizing events to occur at approximately the same time is important to test maximum throughput of the system and is verified by monitoring the message broker (SecutaterityState) queue. This display provides statistics of the queued messages and message rates (e.g, published, delivered, acknowledged) over the past minute. Synchronization is achieved using scripts with a simple TCP client and server to start sending events across all hosts at approximately the same time using *netcat* [51].

4.6 Evaluation Technique

The results are evaluated using R to perform statistical analysis. Event completion time and message delivery time are evaluated using a one-sample t-test and by computing the standard deviation, mean, and 95% confidence interval.

The MDT is comprised of the time taken to send the message from the agent to the broker plus the time for the controller to consume the message. The agent uses the Python *time* library to measure the elapsed time to send and receive a message delivery confirmation from the broker. This value is stored in the *agentData* array until it is sent to the controller to build the data file using the *sendSyncData()* function. As the message is consumed by the controller, the Java *System.nanoTime()* function is used to measure the time the controller takes to receive the message. This time is converted to milliseconds and stored in the node object *consumerDataLog* array.

The ECT is comprised of message delivery time plus the time taken to apply any rules to the firewall using the REST API. The Java *System.nanoTime()* is used to measure the elapsed time between the REST `POST` and `HttpURLConnection 200` response from the firewall. MDT and ECT are defined as:

$$MDT = Time(Agent\ Send) + Time(Controller\ Receive) \quad (2)$$

$$ECT = MDT + Time(Firewall\ REST\ Transaction) \quad (3)$$

The ESR is evaluated by tracking and verifying each flow table update. Reasons for failure include a dropped message or error at the broker, SDN controller, or switch. The component responsible for failure is tracked by noting the last successful step completed during the data collection sequence. Event success rate is further verified by inspection of the running firewall configuration. As shown in Figure 20, the command `curl -X GET http://localhost:8080/wm/firewall/rules/json` returns the current ACL, where the ruleid, source port, destination port, and action are verified.

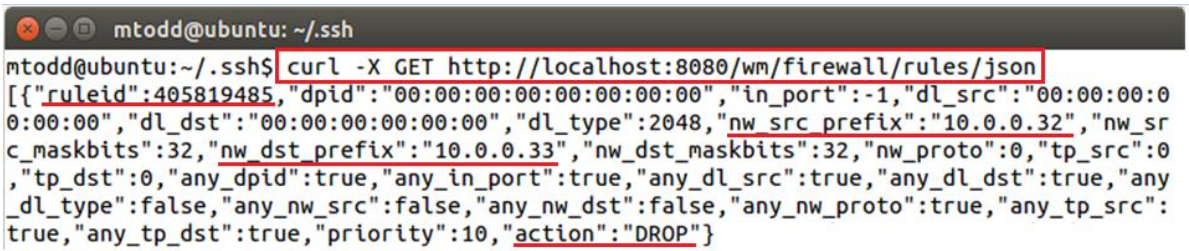


Figure 20. Firewall Access Control List

4.7 Experimental Setup

4.7.1 Overview of Setup.

The test environment, as shown in Figure 21, includes two physical servers and one hardware switch. The servers are both SuperMicro SuperServers 8027R-TRF+ with a Xeon E5-4600 v2, eight 1000BASE-T network interface cards, 12 cores, and 384GB of RAM. The switch is a HP 5900 Series switch (model JG336A)[52] with OpenFlow 1.3 support. Each SuperMicro hosts several guest virtual machines. Windows 7 64-bit (BUILD 6.1.7601) guests are assigned two cores, four gigabytes of RAM, and run the DSCS agent to register

as *nodes* in the DSCS controlled network. The two Ubuntu 14.04 64-bit (Linux 3.13.0-63) hosts are assigned two cores, four gigabytes of RAM, and used for the SDN controller and AMQP broker. All guests have an exclusive 1000BASE-T NIC and all traffic is switched through the HP5900.

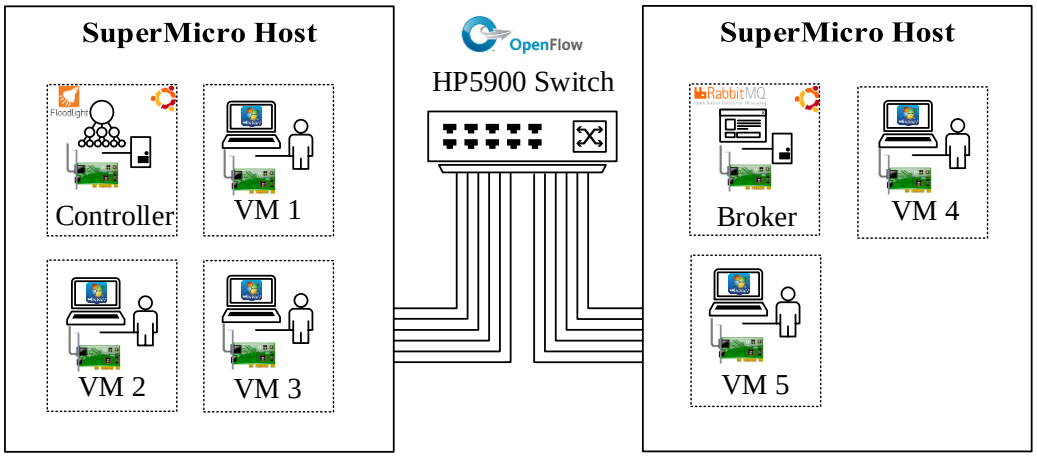


Figure 21. Experiment Environment

4.7.2 Experimental scripts.

In order to start events at approximately the same time, scripts are used to listen for and receive a flat text configuration file containing the number of events to send from each agent. Receipt of this file also notifies the agents to start sending the number of events specified in the transmitted configuration file to the broker.

collect.bat.

- Collet.bat is executed at each agent before start.sh and starts simple netcat connection server listening on port 55555 to receive the test configuration file.

Agent.py.

- Agent.py is executed on each agent after collect.bat and starts the DSCS agent on the host.

start.sh.

- Start.sh is executed on the broker and starts a netcat connection to agents awaiting the test configuration file (named CONFIG).

process_data.sh (n) (m).

- Process_data.sh is executed on the controller and checks the individual data files for correctness (i.e., number of lines and size), and merges all five files into one data file where n is the level (e.g., 1,000, 5,000, 10,000, or 50,000) and m is the trial index.

Test process. Both experiments consist of a set of trails. In each trial, the following steps are followed:

1. Start all agents and listen for the configuration file using *collect.bat* (e.g., `netcat -lp 55555 > start.dat`).
2. Initialize controller and ensure connectivity with switch.
3. Set configuration file with event size parameters.
4. Start collection from broker using *start.sh* (e.g., `netcat 192.168.2.110 55555 < CONFIG`).
5. Merge data into a single file per test using *process_data.sh*.

4.8 Methodology Summary

This chapter describes the methodology used to measure the efficiency and effectiveness of the DSCS framework. The experiments use different workload event levels to determine the operating capacity of the system. Data collection is built into both the agent and SDN controller to account for the time required to send, receive, and process events. Events

are randomly generated at each agent and transferred without delay to the broker and subsequently consumed by the controller. The workload is evenly distributed between the 5 hosts (agents). Within the controller, each registered node tracks events processed by the associated agent to record timing data.

Both experiments use scripts to synchronize and automate initialization and data collection. The number of generated events is varied and multiple trials at each level are performed. Collection, aggregation, and formatting of data is also manged using scripts.

The experiments are evaluated by measuring the time required to send messages to the controller. Since the controller is continually waiting to consume the next available message, possible queuing delay is accounted for in this measurement. Next, the time taken to transfer the event message and take action if required is measured. By combining the message delivery time with the controller processing time, the total event completion time is quantified. Finally, event accountability from end-to-end is monitored and reported as the ratio of successful to total events.

V. Results and Analysis

5.1 Summary of Results

Table 5 contains a summary of results for both experiments. As shown in the top section, Experiment 1 indicates as the event level increases, so too does the event completion time and message delivery time. All levels tested reported less than 5.3 milliseconds from host agent to initiate flow table updates on the controller. Additionally, 100% of the events were successfully handled by the controller. Experiment 2 provides somewhat counterintuitive metrics at first review as seen in the bottom section of Table 5. Event completion time reportedly decreased as the number of events increased, while message delivery time changed very little. However, Experiment 2 is designed to observe the back end of the system (Broker $\rightarrow$ Controller $\rightarrow$ Switch update). This is achieved by filling the queue with messages before initiating the controller (consumer) thereby reducing consumer message time significantly. The first cause for this reduced ECT is that the controller retrieves messages from the broker as quickly as possible. This eliminates the delay introduced by the peak message rate of each host being 40 msg./sec.

The second reason is the controller update time during system startup. Until a stable system state is reached (approximately 3,000 events), it is difficult to report an accurate steady running state. To better represent this mean, Table 6 provides statistics excluding the first 3,000 events highlighted in Figure 22.

Table 5. Experiment 1 and 2 - Summary of Results

Events	Experiment	ECT (ms)	MDT(ms)	ESR	95% conf.(ECT)	p-value
1,000	1	4.09	1.64	100%	4.06 - 4.13	$< 2.2\ e^{-16}$
5,000	1	4.73	2.98	100%	4.70 - 4.76	$< 2.2\ e^{-16}$
10,000	1	4.98	3.39	100%	4.95 - 5.01	$< 2.2\ e^{-16}$
50,000	1	5.27	3.88	100%	5.25 - 5.29	$< 2.2\ e^{-16}$
10,000	2	3.08	1.46	100%	3.06 - 3.08	$< 2.2\ e^{-16}$
50,000	2	2.89	1.54	100%	2.87 - 2.90	$< 2.2\ e^{-16}$

Table 6. Experiment 2 - Summary Excluding First 3K Events

Events	Experiment	ECT (ms)	MDT (ms)	ESR	95% conf.(ECT)	p-value
7,000	2	2.86	1.45	100%	2.85 - 2.87	$< 2.2\ e^{-16}$
47,000	2	2.84	1.54	100%	2.82 - 2.86	$< 2.2\ e^{-16}$

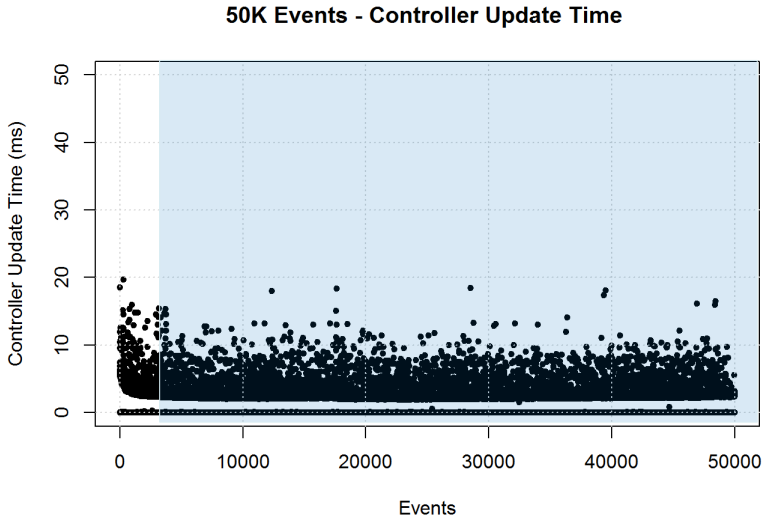


Figure 22. Steady System State

The solid line formed near zero across the bottom of the graph in Figure 22 are observations that require little action from the controller and therefore near zero processing time. These events cause the mean controller update time to decrease as the treatment level increases.

Figure 23 plots in red the four ECT points provided in Table 5. By fitting a linear model of y as function of $log(x)$, the black line suggests a logarithmic relationship between the number of events and ECT. Using this model, the following equations are derived using R. (R code in Appendix C, Figures 41 and 43):

$$y = 0.30343 \log(x) + 4.17398 \tag{4}$$

$$y_{[95\%\ Upper]} = 0.495449 * log(x) + 4.636387 \tag{5}$$

$$y_{[95\%\ Lower]} = 0.111413 * log(x) + 3.711582 \tag{6}$$

To test (4), a single run at 100,000 events is performed resulting in a 5.23 ECT. The model in (4) at $x = 100$ (in thousands of events) returns a 5.57 ECT. The single run 5.23 ECT is within the predicted 95% interval of 4.22 to 6.92 milliseconds and shown as '*' in Figure 23. Therefore, the model successfully predicted the result within 95% confidence. Additionally, the R^2 value of 0.96 indicates 96% of the variance in the test data is explained by the model.

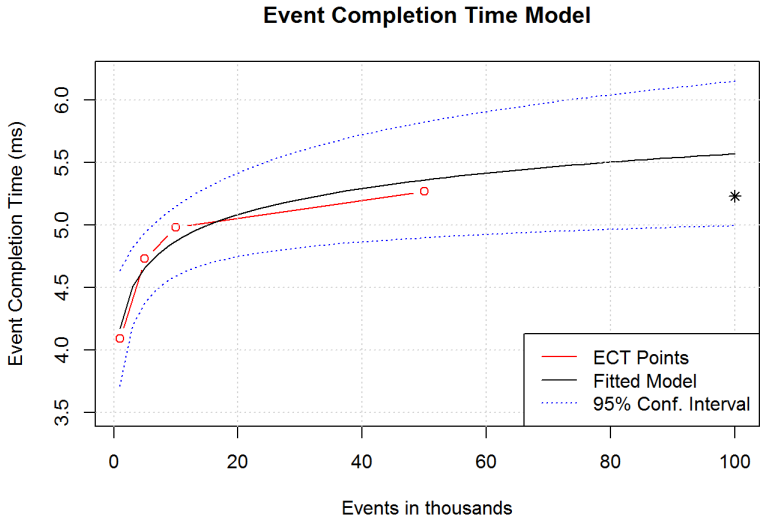


Figure 23. Experiment 1 - Event Completion Time Model

Figure 24 shows MDT also performs logarithmically. Since ECT includes MDT and controller update time (found to stay relatively static), the logarithmic performance of ECT is contributed to MDT.

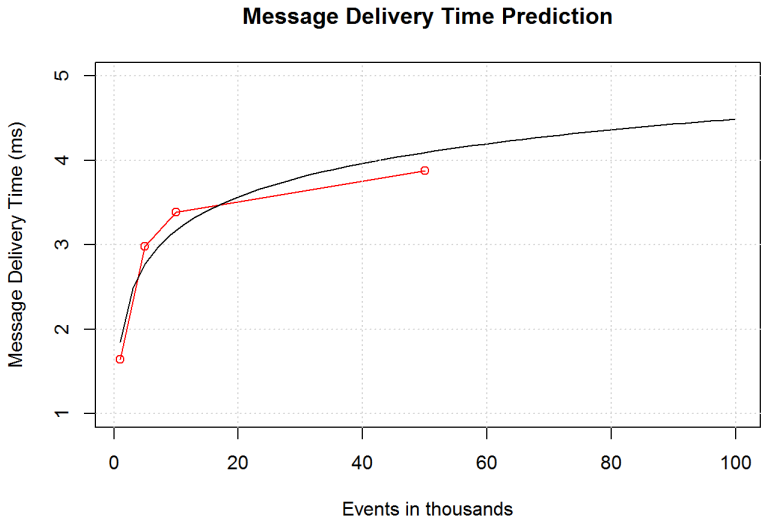


Figure 24. Experiment 1 - Message Delivery Time Prediction

5.2 Experiment 1: DSCS Full System

Table 5 in Section 5.1 provided a high-level summary of results for Experiment 1. The following figures and analysis present a more detailed look at the collected data. This data and the associated R code is found in Appendix C.

5.2.1 System Performance and Efficiency.

Table 7 summarizes mean elapsed time for each level tested and associated effective message rates. As expected, the elapsed time is approximately linear. However, the effective message rates reached a peak of 195 msg./sec. and did not increase based on treatment level. Upon closer inspection of RabbitMQ logs as seen in Appendix B for all levels (1,000-50,000), the maximum publish and consume rates peak at approximately 200 msg./sec., resulting in each host sending an average of ($200 \frac{msg}{sec} \div 5\ hosts$) 40 msg./sec. It is important to remember the DSCS system is designed to send messages (events) based on configured security triggers. Therefore if more than a few msg./sec. per host are triggered, network administrators would be notified immediately.

Table 7. Experiment 1 - Trials Elapsed Time (sec)

Level	Mean	Msg./Sec.	Std Dev	Min	Median	Max
1,000	5.43	183.85	0.15	5.15	5.42	5.78
5,000	25.77	193.99	0.25	25.22	25.77	26.27
10,000	51.44	194.40	0.43	50.57	51.36	52.50
50,000	255.50	195.69	1.49	252.8	255.3	258.5

5.2.2 Message Delivery Time.

MDT is the first metric directly observed consisting of both agent and consumer message time, e.g., the time it takes to send a message from the agent to the broker, and then from the broker to the consumer respectively. The following sections discuss these two components in detail.

5.2.2.1 Agent Message Time.

Agent message time is relatively static and does not change as the level of events increases. As determined above, irrespective of the number of events, each host sends 40 msg/sec. Table 8 and Figure 25 present how the message time is not impacted by the number of events flowing through the system by eventually reaching 1.38 milliseconds.

Table 8. Experiment 1 - Agent Message Time (ms)

Level	Mean	Std Dev	Min	Median	Max	95% Conf	p-value
1,000	1.41	0.88	1.35	1.41	1.49	1.40 - 1.42	$< 2.2\ e^{-16}$
5,000	1.40	0.02	1.35	1.40	1.46	1.39 - 1.40	$< 2.2\ e^{-16}$
10,000	1.39	0.02	1.36	1.40	1.46	1.39 - 1.40	$< 2.2\ e^{-16}$
50,000	1.38	0.01	1.36	1.39	1.40	1.38 - 1.39	$< 2.2\ e^{-16}$

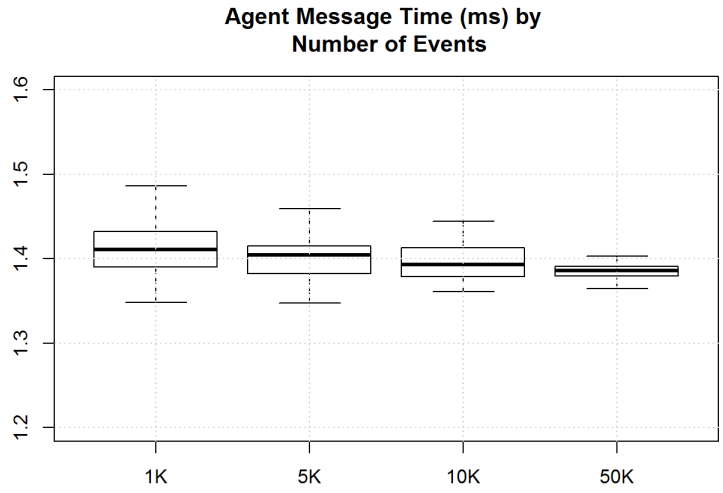


Figure 25. Experiment 1 - Agent Message Time Summary

5.2.2.2 Consumer Message Time.

Unlike agent message time, consumer message time, or the time taken for the SDN controller to retrieve a message from the broker, does increase event level. Table 9 shows an approximate logarithmic increase in the mean consumer message time. This growth occurs as the number of events increase and each host reaches the maximum message rate of approximately 40 msg/sec. Additionally, it is believed this increase is caused by the sustained processing of events as the controller module competes for resources from the OS.

Table 9. Experiment 1 - Consumer Message Time (ms)

Level	Mean	Std Dev	Min	Median	Max	95% Conf	p-value
1,000	0.22	0.03	0.17	0.23	0.35	0.21 - 0.24	$< 2.2\ e^{-16}$
5,000	1.58	0.08	1.33	1.61	1.72	1.54 - 1.61	$< 2.2\ e^{-16}$
10,000	2.00	0.10	1.80	2.00	2.20	1.96 - 2.04	$< 2.2\ e^{-16}$
50,000	2.50	0.04	2.38	2.49	2.59	2.47 - 2.51	$< 2.2\ e^{-16}$

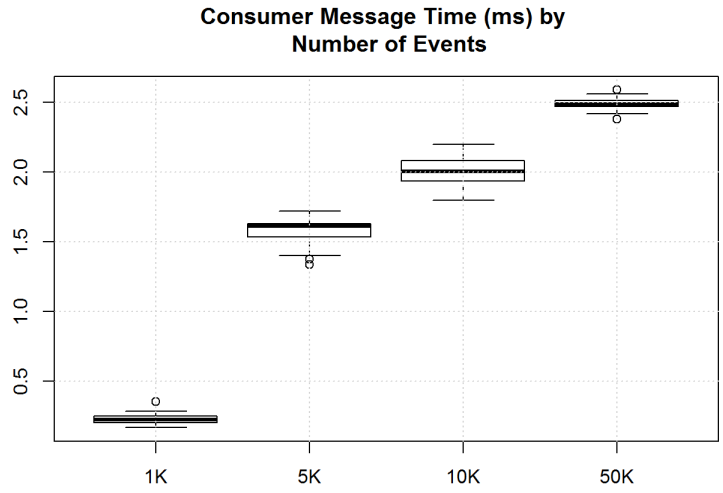


Figure 26. Experiment 1 - Mean Consumer Message Time Summary

5.2.2.3 MDT Summary.

MDT is the sum of both agent and consumer message times. As shown in Table 8, agent message time is relatively static and not directly impacted by the treatment. However, Table 10 and Figure 27 show how consumer message time responds as the controller is required to process more events over time. MDT grows even though the message rate of approximately 200 msg/sec does not increase. It is believed this increase in time occurs as the controller must compete for system resources to process the growing number of events.

Table 10. Experiment 1 - Message Delivery Time (ms)

Level	Mean	Std Dev	Min	Median	Max	95% Conf	p-value
1,000	1.64	0.04	1.55	1.63	1.73	1.62 - 1.66	$< 2.2\ e^{-16}$
5,000	2.97	0.10	2.68	2.99	3.18	2.94 - 3.02	$< 2.2\ e^{-16}$
10,000	3.39	0.12	3.17	3.41	3.65	3.35 - 3.44	$< 2.2\ e^{-16}$
50,000	3.88	0.04	3.77	3.87	3.99	3.86 - 3.89	$< 2.2\ e^{-16}$

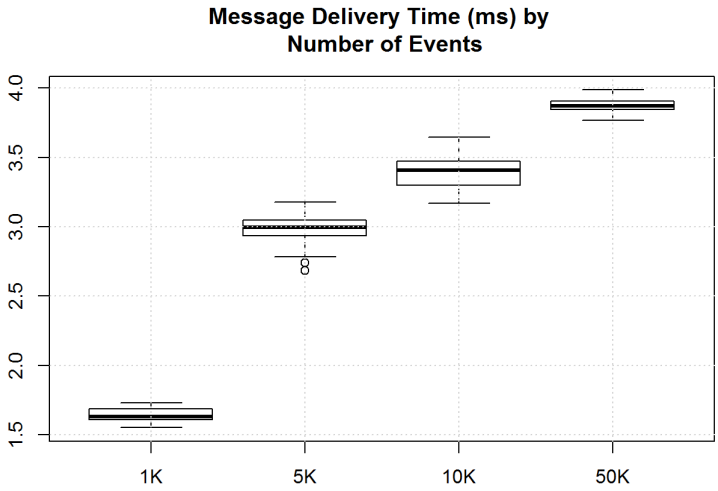


Figure 27. Experiment 1 - Mean Message Delivery Time Summary

5.2.3 Event Success Rate.

ESR is the next metric observed during testing and is a ratio of successful to total events processed. Across all trails every event was processed successfully resulting in a 100% ESR for all levels. During these observations, the level of events did not affect updates performed by the controller. ESR would have been impacted if the system experienced a critical fault.

5.2.4 Event Completion Time.

ECT is the final metric observed during testing and is the elapsed time required to trigger, communicate, and apply flow table updates at the controller. As described by (3) in Section 4.6, ECT is the sum of MDT and firewall REST transaction time.

5.2.4.1 Controller Update Time.

Controller update time consists of controller processing and firewall REST transactions and varies the most at system startup. Figure 28 illustrates how at 10,000 events approximately the first third of the observations are trending downward towards the steady system state. This leveling is observed only in controller update time and as the controller has not yet reached a normal operating state, e.g., nodes are created and added to the network

data structure. In this experiment the complete dataset to include the ramp-up period is included when calculating the statistics in Table 11 and Figure 29. Treatment level 1,000 is much higher because the startup period represents a larger proportion of the total treatment. However, if the ramp-up period is ignored, the mean would be relatively static and not respond to the treatment level as demonstrated in Experiment 2.

Table 11. Experiment 1 - Controller Update Time (ms)

Level	Mean	Std Dev	Min	Median	Max	95% Conf	p-value
1,000	2.45	0.09	2.27	2.44	2.63	2.42 - 2.49	$< 2.2\ e^{-16}$
5,000	1.75	0.04	1.7	1.75	1.83	1.74 - 1.77	$< 2.2\ e^{-16}$
10,000	1.59	0.05	1.52	1.58	1.69	1.57 - 1.60	$< 2.2\ e^{-16}$
50,000	1.39	0.01	1.36	1.39	1.43	1.39 - 1.40	$< 2.2\ e^{-16}$

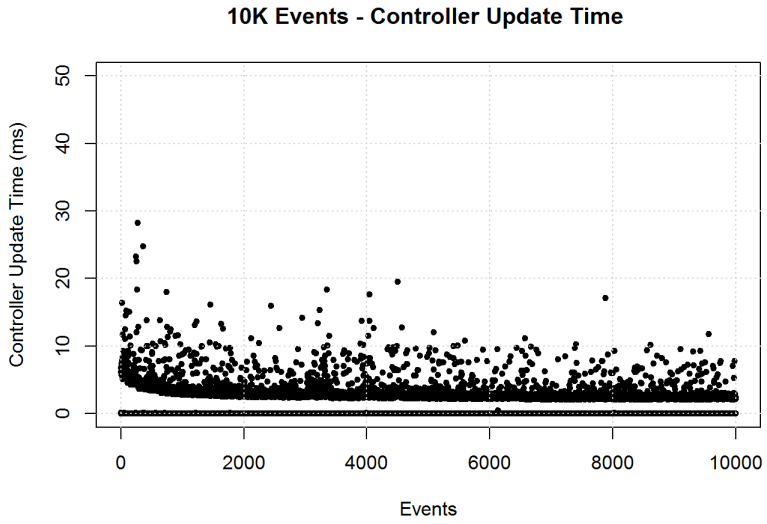


Figure 28. Experiment 1 - Leveling Controller Update Time

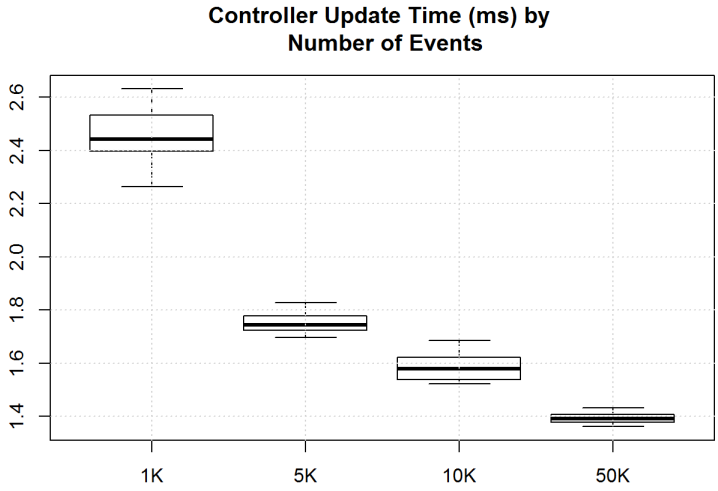


Figure 29. Experiment 1 - Mean Controller Update Time Summary

5.2.4.2 ECT Summary.

ECT is the sum of MDT and controller update time for each event respectively. As shown in Table 10, MDT increases along with the treatment level primarily due to consumer message time. However, controller update time tends towards a mean of approximately 1.42 milliseconds across all treatment levels indicating a relatively static result for all values tested. Table 12 and Figure 30 demonstrate how ECT does increase per treatment level as expected.

Table 12. Experiment 1 - Event Completion Time (ms)

Level	Mean	Std Dev	Min	Median	Max	95% Conf	p-value
1,000	4.09	0.09	3.94	4.08	4.33	4.06 - 4.12	$< 2.2\ e^{-16}$
5,000	4.73	0.08	4.51	4.74	4.90	4.70 - 4.76	$< 2.2\ e^{-16}$
10,000	4.98	0.08	4.84	4.98	5.18	4.95 - 5.01	$< 2.2\ e^{-16}$
50,000	5.27	0.04	5.20	5.27	5.36	5.25 - 5.29	$< 2.2\ e^{-16}$

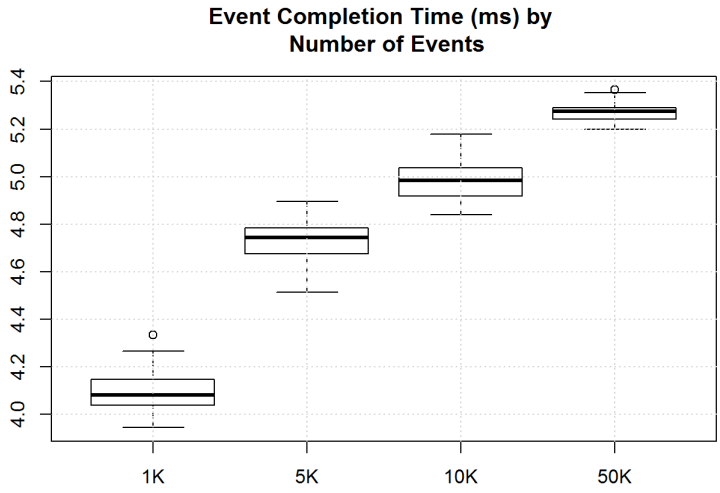


Figure 30. Experiment 1 - Mean Event Completion Time Summary

5.3 Experiment 2: Back End (Preload Broker)

Unlike Experiment 1, this test attempts to analyze the back end of the DSCS system for two reasons. First, the test environment does not support enough hosts to test the maximum consume rate of the controller. Learning the maximum consume rate will provide insight into how many hosts this environment can support. Next is to test the system at a higher rate than 200 events/sec (the maximum Experiment 1 is capable of performing). The results from this experiment provide data to help explain if the DSCS system could scale in a larger environment. While the configuration did not change from Experiment 1, the broker is preloaded with events *before* enabling the controller (consumer).

5.3.1 System Performance and Efficiency.

All reported statistics for Experiment 2 include events after the first 3,000 where approximately a steady running state is achieved. The following figures and analysis present a closer look at the collected data. Appendix D provides additional detail and associated R code. Table 13 shows mean elapsed time for each level tested and associated effective message rates.

Like Experiment 1, the time required to perform the tests is approximately linear. However, the effective message rate reached a peak of 380 msg./sec. and did increase based on treatment level. Referencing the RabbitMQ logs in Appendix D for both 10,000 and 50,000, the maximum consume rate reached 600 msg./sec. This is a drastic increase from 200 msg./sec. in Experiment 1. This reveals the broker in this configuration could process from one msg./sec. (380 hosts) to 40 msg./sec. (9 hosts) without delay from hosts to controller. It is important to note that in a production environment events may not occur every second or even every minute.

Table 13. Experiment 2 - Trial Elapsed Time (sec)

Level	Mean	Msg./Sec.Rate	Std Dev	Min	Median	Max
10,000	33	303.10	1.55	30.00	32.95	40.00
50,000	131.3	380.68	1.61	129.10	130.90	136.50

5.3.2 Message Delivery Time.

MDT for Experiment 2 still consists of agent message time and consumer message time, but should not be directly compared to results in Experiment 1 since messages are preloaded on the broker. This results in a relatively static MDT even as the treatment level increases.

5.3.2.1 Agent Message Time.

Similar to Experiment 1, agent message time is relatively static and does not significantly change as the treatment level increases. Table 14 and Figure 31 present statistics and show agent message time ranges between 1.28 to 1.44 milliseconds.

Table 14. Experiment 2 - Agent Message Time (ms)

Level	Mean	Std Dev	Min	Median	Max	95% Conf	p-value
10,000	1.33	0.02	1.28	1.33	1.39	1.32 - 1.34	$< 2.2\ e^{-16}$
50,000	1.39	0.04	1.31	1.41	1.44	1.37 - 1.40	$< 2.2\ e^{-16}$

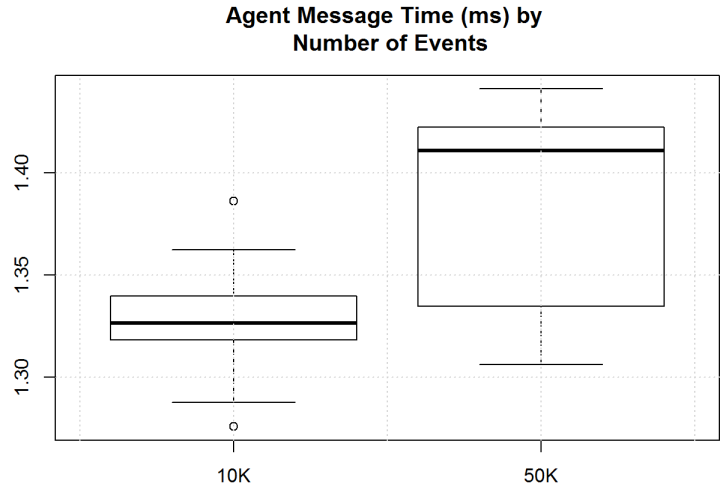


Figure 31. Experiment 2 - Mean Agent Message Time Summary

5.3.2.2 Consumer Message Time.

Since the controller is not required to wait on agents to send events, consumer message time does not increase significantly with treatment level. This is expected as all messages are preloaded before the controller is started. Table 15 and Figure 32 show this slight increase. Unlike Experiment 1, the controller does not have to wait on agents to send messages. The slight increase of mean is believed to be caused by read time from retrieving messages stored on disk within the broker.

Table 15. Experiment 2 - Consumer Message Time (ms)

Level	Mean	Std Dev	Min	Median	Max	95% Conf	p-value
10,000	0.13	0.01	0.11	0.13	0.15	0.12 - 0.13	$< 2.2\ e^{-16}$
50,000	0.16	0.01	0.14	0.16	0.17	0.15 - 0.16	$< 2.2\ e^{-16}$

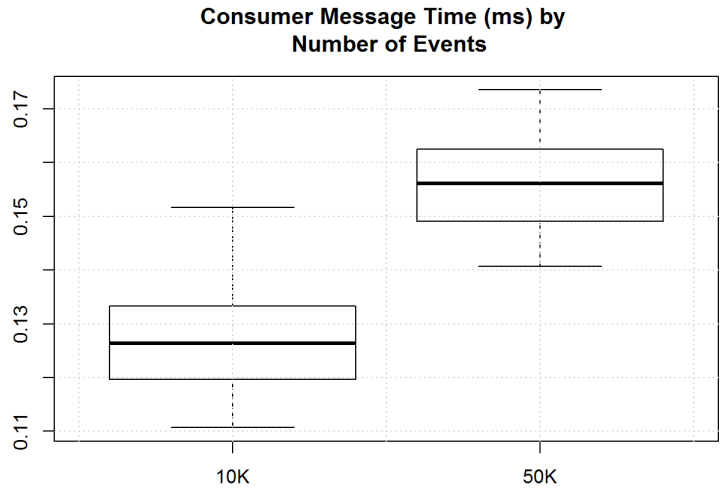


Figure 32. Experiment 2 - Mean Consumer Message Time Summary

5.3.2.3 MDT Summary.

Table 16 and Figure 33 show MDT does slightly increase but is not as significantly as Experiment 1. Since the system is preloaded with messages, the consumer is not forced to wait for messages to arrive in the broker queue. Therefore the small increase is attributed to the number of messages stored on disk and associated access time.

Table 16. Experiment 2 - Message Delivery Time (ms)

Level	Mean	Std Dev	Min	Median	Max	95% Conf	p-value
10,000	1.45	0.02	1.40	1.46	1.52	1.44 - 1.46	$< 2.2\ e^{-16}$
50,000	1.54	0.05	1.45	1.56	1.60	1.53 - 1.56	$< 2.2\ e^{-16}$

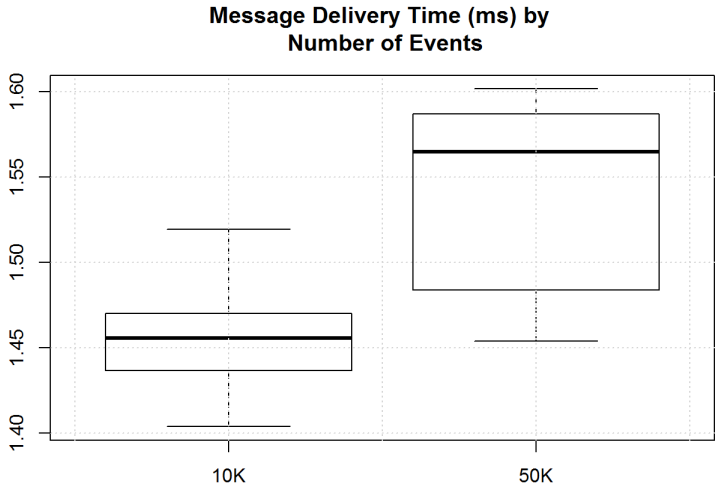


Figure 33. Experiment 2 - Mean Message Delivery Time Summary

5.3.3 Event Success Rate.

ESR for this experiment is 100%. This means as the controller (consumer) works at the peak speed of 380 msg./sec., every event is processed from agent to flow table update successfully.

5.3.4 Event Completion Time.

Much like Experiment 1, ECT decreases as the treatment level increase but is trending towards the same mean. This behavior is expected as many events do not require a firewall modification and therefore have a near zero processing time.

5.3.4.1 Controller Update Time.

Though the two trials are statisticaly different as seen in Figure 34 and Table 17, the mean time to update the controller is less than 1.5 milliseconds. This decrease in update time, though statistically different, is believed to converge at the same mean as discussed in Section 5.1.

Table 17. Experiment 2 - Controller Update Time (ms)

Level	Mean	Std Dev	Min	Median	Max	95% Conf	p-value
10,000	1.41	0.03	1.35	1.41	1.47	1.39 - 1.42	$< 2.2\ e^{-16}$
50,000	1.30	0.01	1.26	1.29	1.34	1.29 - 1.30	$< 2.2\ e^{-16}$

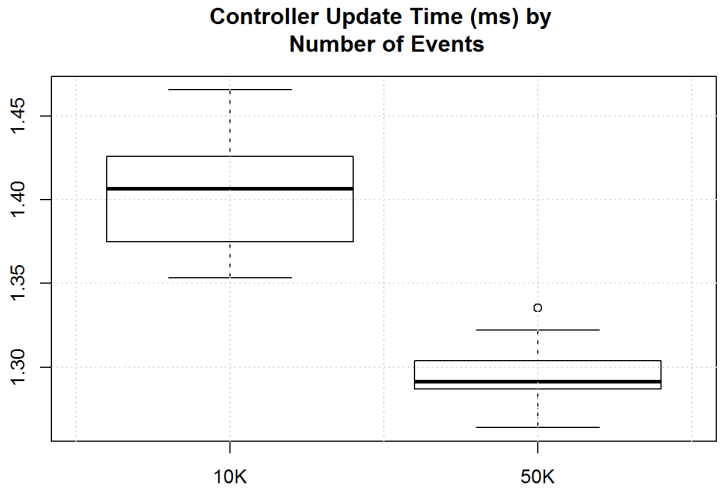


Figure 34. Experiment 2 - Mean Controller Update Time Summary

5.3.4.2 ECT Summary.

ECT for both treatment levels are very similar. Even though message delivery time increases with treatment level, consumer update time decreases by approximately the same amount (0.10 millisecond). This occurs as the mean controller update time continues to decrease and more events with less processing time are encountered. Unlike Experiment 1, since MDT did not increase significantly (due to preloaded broker), this behavior is expected.

Table 18. Experiment 2 - Event Completion Time (ms)

Level	Mean	Std Dev	Min	Median	Max	95% Conf	p-value
10,000	2.86	0.04	2.80	2.86	2.94	2.85 - 2.87	$< 2.2\ e^{-16}$
50,000	2.84	0.05	2.74	2.86	2.90	2.82 - 2.86	$< 2.2\ e^{-16}$

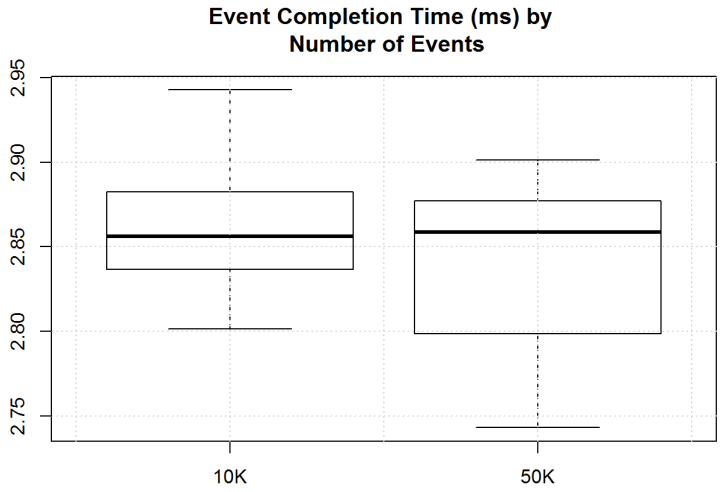


Figure 35. Experiment 2 - Mean Event Completion Time Summary

VI. Conclusions and Recommendations

6.1 Introduction

This chapter summarizes the research performed. Section 6.2 presents the conclusions reached during experimentation. Section 6.3 discusses the impact of this research. Section 6.4 presents potential future work.

6.2 Research Conclusions

This research is successful in achieving the goal of developing the DSCS framework to manage layer-2 flow tables initiated from host-based events. As hypothesized, event completion time increased with the number of events and event success rate was 100%. However, message delivery time did not stay the same as expected and is the main factor for additional delay as event level increases. Using the design in Chapter 3 and goals in Section 4.1, the framework is tested and found to satisfy the desired efficiency, effectiveness, and ease of implementation.

6.2.1 Efficiency.

Results show flow table updates are made for all tested levels in less than 5.27 milliseconds. This indicates near real-time modification of flow tables which is well below the stated goal of one second (Section 1.3). Additionally, as the number of events processed increases from 1,000 to 50,000, the design scales logarithmically due to MDT and is limited primarily by the message rate of the agent (40 msg./sec.). Experiment 2 provides insight into the maximum effective consume rate of the controller at 380 msg./sec. (e.g., one controller can support up to 380 hosts at one msg./sec.).

6.2.2 Effectiveness.

Every event is successfully processed for both experiments resulting in a 100% ESR. Since the broker uses authentication and SSL when messaging, the contents are protected

from snooping. While the critical route and flow based profiles features were not directly tested in either experiment, all routes added are verified. Each firewall rule is applied immediately except for those with active connections.

6.2.3 Ease of implementation.

As demonstrated in both experiments, AMQP messaging is platform independent by design and available in most mainstream operating systems. Windows 7 is used in this research to support the use of DSCS in the most common of environments. Little configuration is required on the host agent, only requiring authentication credentials and custom modules to interface with existing security software (anti-virus, HBSS, etc.).

6.3 Significance of Research

6.3.1 Contributions.

The main drive for this research is to take advantage of security event information at endpoints and propagate this awareness to forwarding devices using SDN. This research provides three contributions unique to DSCS. First, it applies network protection across layer-2 devices based on host-level security events. This approach helps mitigate the chance of an adversary moving laterally within a network once a compromise is discovered. Next, DSCS uses host security index tracking within the SDN controller to monitor individual systems and notify network administrators when appropriate. Finally, this system allows for customized flow-based profiles to ensure critical routes are available to avoid mission interruption when security events occur. Critical routes allow for a more targeted response instead of blocking all host communications.

6.3.2 Applications.

DSCS is appropriate for any environment with the desire to bolster network defenses with information gathered from hosts. However, this framework was designed for organizations with many hosts and locations in mind.

6.4 Future Work

While there are many possibilities for additional research within the DSCS system, the following are particularly interesting because of the impact on incident recovery and response. The first and most extensive is to incorporate a database to allow for multiple networks or locations to share event information and push flow table updates to all switches, allowing DSCS to scale to support an entire enterprise.

Another feature is to incorporate system artifact collection into the agent. Artifacts such as system logs (event, application, software, security, etc.), memory capture, and IP traffic capturing. These logs can be preserved and transfered to an analysis platform to speed up response and recovery actions. After automated analysis using a tool such as Anubis [53], the results can be parsed and malicious addresses added as a deny flow table entry.

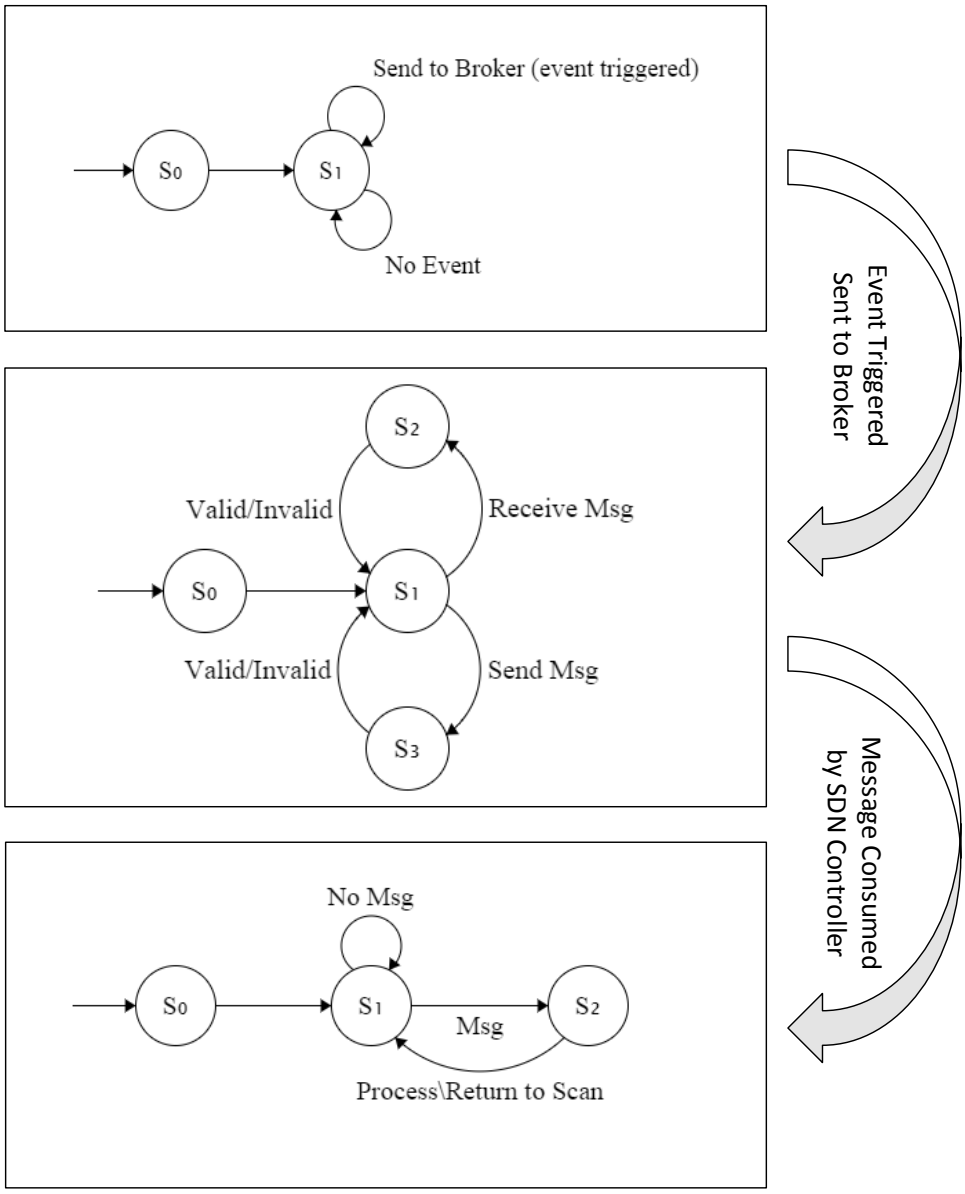
A key limitation of this research is the Floodlight Stateless Firewall used during development could not interrupt active connections even if they were found to be malicious. Future work could address this problem by modifying the agent to kill all active connections to malicious addresses before sending a block event. Another approach is to use the controller to interrupt active connections before updating the flow table.

The DSCS system should also be tested in an environment where the agent actually interfaces with a detection system. This can be accomplished using the same test environment as this research. Next an interface between detection software (anti-virus, HBSS, etc.) and the agent is required. The agent must then be configured to trigger based on detection system flags. Finally, security policy on the victim systems must be violated in such a way the detection system identifies the attack.

6.5 Chapter Summary

In summary, this chapter presents the research conclusions and significance for this research. Additionally, the applications and ideas for future work to expand on the DSCS framework is discussed.

Appendix A. Inter-component State Diagram



Appendix B. Power Test for Sample Size

```
Power test for one trial at 1000 samples, for Event Completion Time in Experiment 1.
Event Completion Time (the greatest variance of all metrics)
> var(1000_r1$`Event Completion Time`)
[1] 13.01189
> sd(1000_r1$`Event Completion Time`)
[1] 3.607199

> power.t.test(n = NULL, delta = 1, power = .90, sd = 3.607199)

     Two-sample t test power calculation

              n = 274.4067
          delta = 1
             sd = 3.607199
      sig.level = 0.05
          power = 0.9
    alternative = two.sided
```

```
Power test for one trial at 10000 samples, for Event Completion Time in Experiment 2.
Event Completion Time (the greatest variance of all metrics)

> var(r1_10000F2$total)
[1] 13.34055
> sd(r1_10000F2$total)
[1] 3.652472

> power.t.test(n = NULL, delta = 1, power = .90, sd = 3.652472)

     Two-sample t test power calculation

              n = 281.3135
          delta = 1
             sd = 3.652472
      sig.level = 0.05
          power = 0.9
    alternative = two.sided
```

Appendix C. Experiment 1 Graphs and Supporting R Code

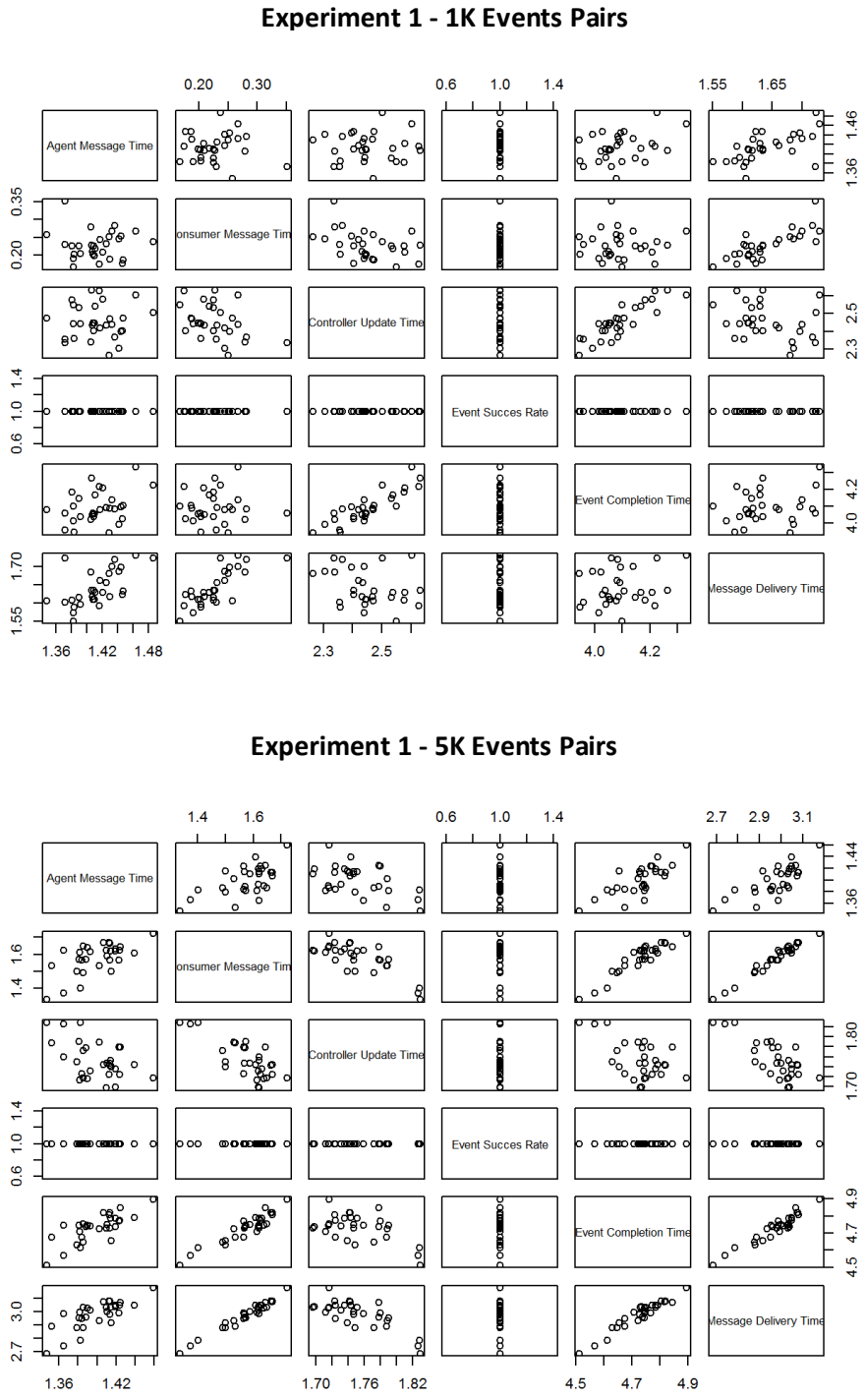
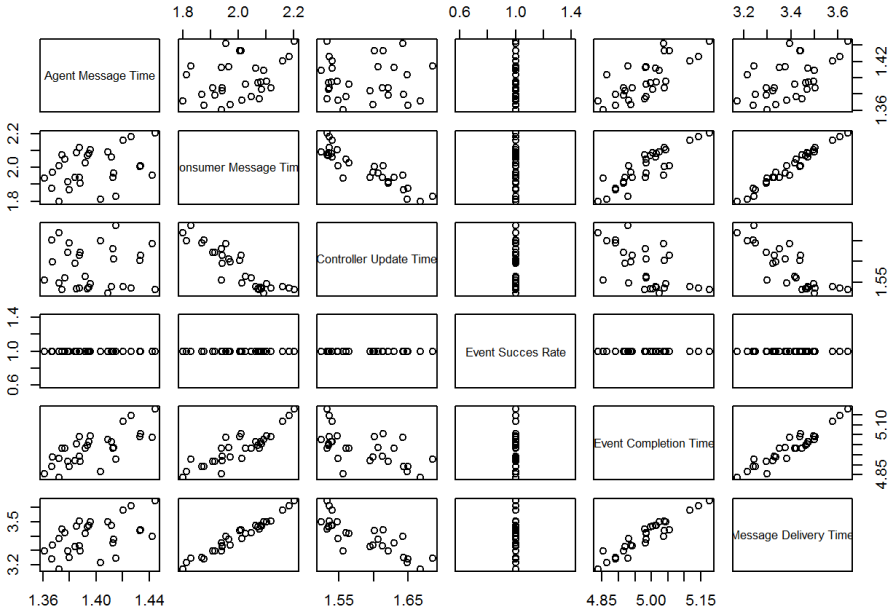


Figure 36. Experiment 1 - Pairs Plot 1K and 5K

Experiment 1 - 10K Events Pairs



Experiment 1 - 50K Events Pairs

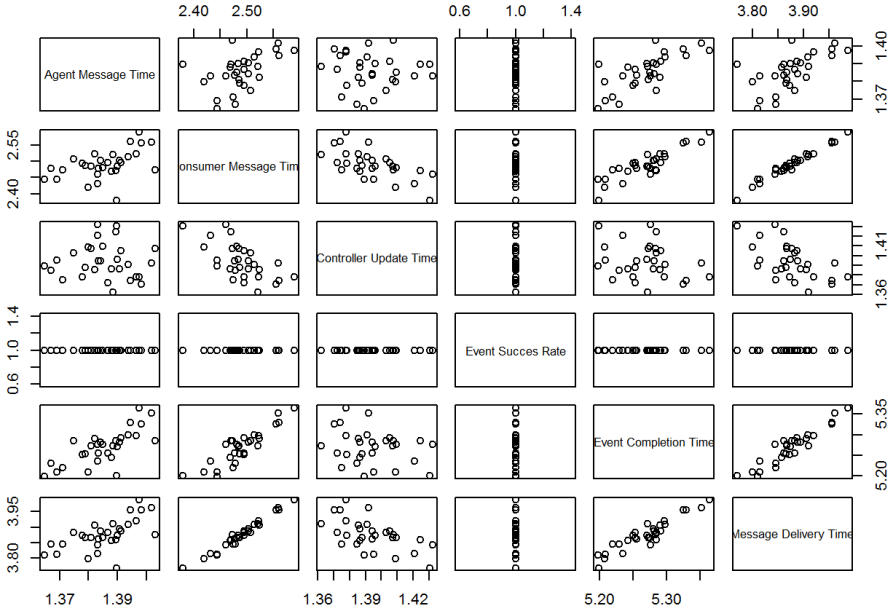


Figure 37. Experiment 1 - Pairs Plot 10K and 50K

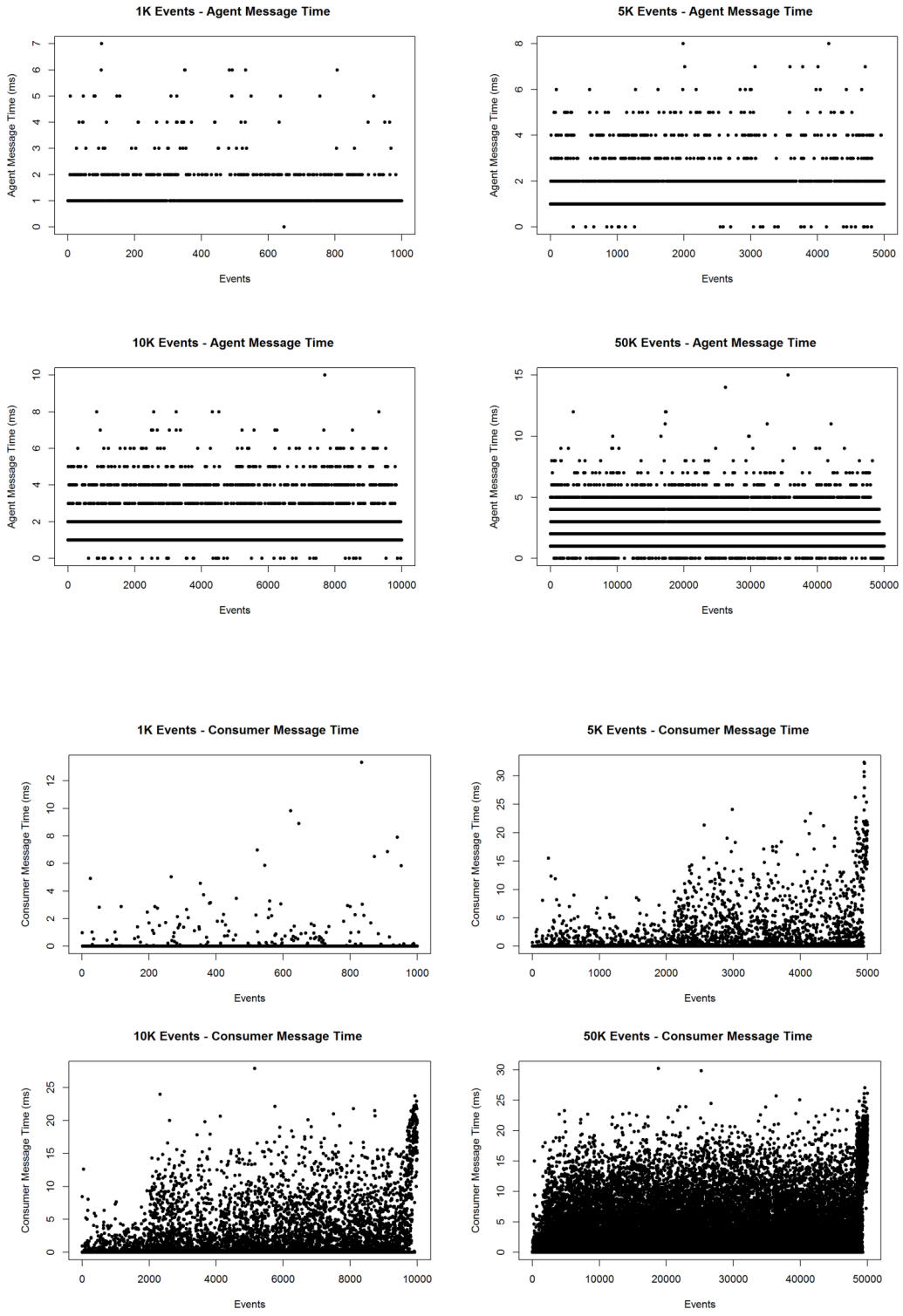


Figure 38. Experiment 1 - Agent Message Time and Consumer Message Time Data Plot

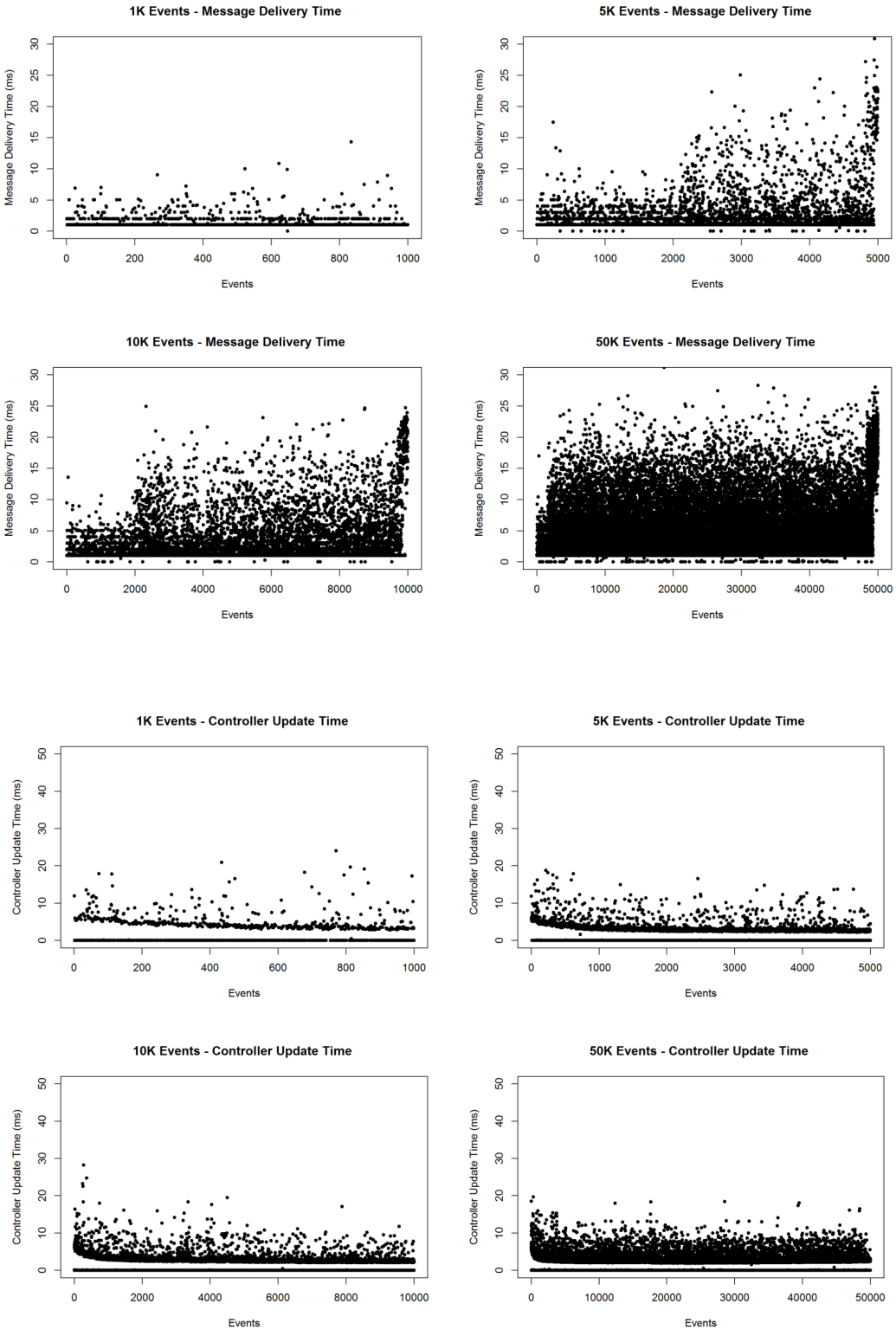


Figure 39. Experiment 1 - Message Delivery Time and Controller Update Time Data Plot

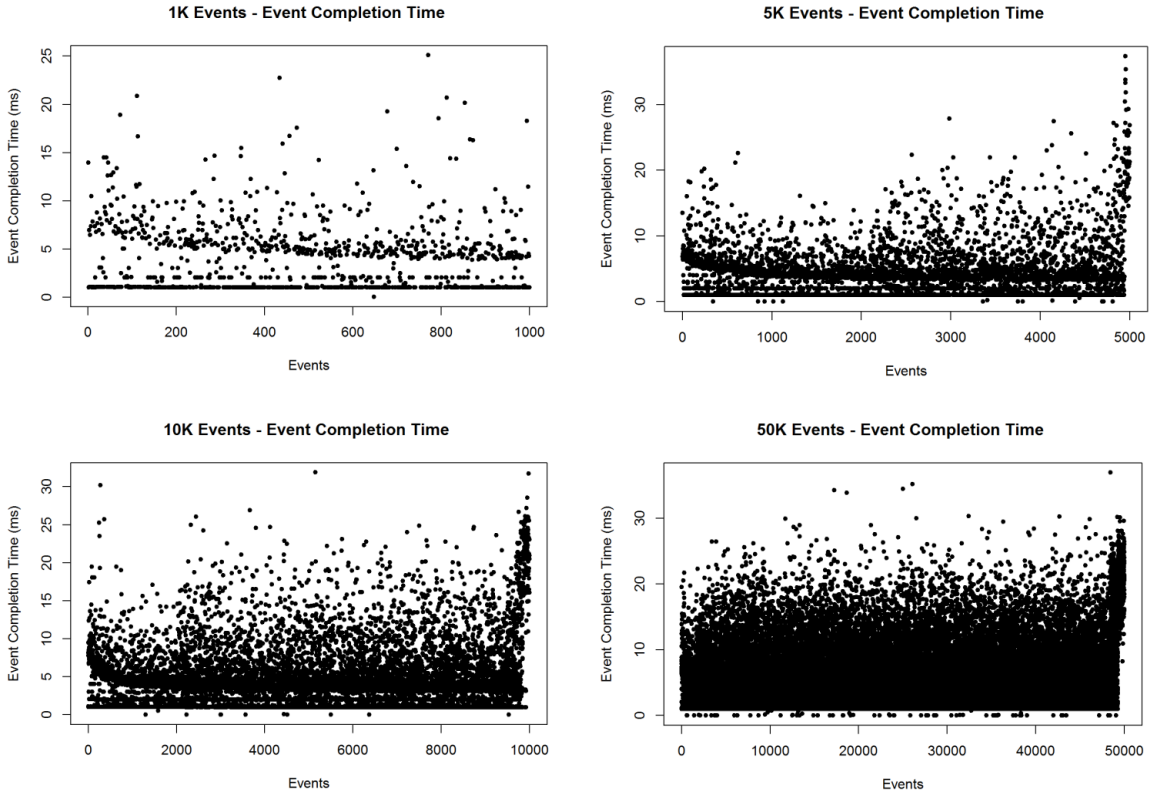


Figure 40. Experiment 1 - Event Completion Time Data Plot

```
# Values from Experiment 1, Table 8 ECT 1-50K
x = c(1, 5, 10, 50)
y = c(4.09, 4.73, 4.98, 5.27)
# Linear model for y as a function of log(x)
fit <- lm(y~log(x))
# Sequence used to draw line, populated by predict function below
xx <- seq(1,100, length=50)
# Plot of ECT data from Table 8 (red)
plot(x,y, type="b", col="red", xlim=c(1, 100), ylim=c(3.5, 6.2), main="Event
    ↪ Completion Time Model", xlab = "Events in thousands", ylab="Event Completion
    ↪ Time (ms)", lwd=1)
# Predicted using fit of y~log(x)
lines(xx, predict(fit, data.frame(x=xx), col="blue"), lwd=1, lty=1)
conf.lines = predict(fit, data.frame(x=xx), interval = "confidence")
lines(xx, conf.lines[,2], lty=3, col="blue")
lines(xx, conf.lines[,3], lty=3, col="blue")
> summary(fit)
Call:
lm(formula = y ~ log(x))
Residuals:
       1        2        3        4
-0.08398  0.06766  0.10734 -0.09102
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  4.17398    0.10747  38.839 0.000662 ***
log(x)       0.30343    0.04463   6.799 0.020954 *
---
Signif. codes:  0    ***    0.001    **    0.01    *    0.05    .    0.1         1
Residual standard error: 0.1254 on 2 degrees of freedom
Multiple R-squared:  0.9585,    Adjusted R-squared:  0.9378
F-statistic: 46.23 on 1 and 2 DF,  p-value: 0.02095
#y-predicted values in equation form y=mx+b, where x = log(x)
> 0.30343 * log(60) + 4.17398
[1] 5.416327
> 0.30343 * log(70) + 4.17398
[1] 5.463101
> 0.30343 * log(80) + 4.17398
[1] 5.503618
> 0.30343 * log(100) + 4.17398
[1] 5.571327
```

Figure 41. R Code - Linear Model Fit Log(x) ECT

```
# Values from Table 8 MDT 1-50K
x = c(1, 5, 10, 50)
y = c(1.64, 2.98, 3.39, 3.88)
# Linear model for y as a function of log(x)
fit <- lm(y~log(x))
# Sequence used to draw line, populated by predict function below
xx <- seq(1,100, length=50)
# Plot of ECT data from Table 8 (red)
plot(x,y, type="o", col="red", xlim=c(1, 100), ylim=c(1, 5), main="Message Delivery
    ↪ Time Prediction", xlab = "Events in thousands", ylab="Message Delivery Time (
    ↪ ms)")
#error.bars(x, add)
# Predicted using fit of y~log(x)
lines(xx, predict(fit, data.frame(x=xx)), col="blue"))
> summary(fit)
Call:
lm(formula = y ~ log(x))
Residuals:
      1       2       3       4
-0.2114  0.2061  0.2189 -0.2136
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   1.8514     0.2577   7.185   0.0188 *
log(x)        0.5732     0.1070   5.357   0.0331 *
---
Signif. codes:  0    ***    0.001    **    0.01    *    0.05    .    0.1         1
Residual standard error: 0.3006 on 2 degrees of freedom
Multiple R-squared:  0.9348,    Adjusted R-squared:  0.9023
F-statistic:  28.7 on 1 and 2 DF,  p-value: 0.03313
> 0.5732 * log(60) + 1.8514
[1] 4.198278
> 0.5732 * log(70) + 1.8514
[1] 4.286637
> 0.5732 * log(80) + 1.8514
[1] 4.363178
> 0.5732 * log(100) + 1.8514
[1] 4.491084
>
```

Figure 42. R Code - Linear Model Fit Log(x) MDT

```
# 95% Confidence for Linear model: y ~ log(x)
fit <- lm(y~log(x))
> fit

Call:
lm(formula = y ~ log(x))

Coefficients:
(Intercept)        log(x)
     4.17398        0.30343

> confint(fit, level = 0.95)
                2.5 %    97.5 %
(Intercept) 3.7115828 4.6363870
log(x)      0.1114132 0.4954494
# Lower
> 0.11141 * log(100) + 3.71158
[1] 4.22466
# Upper
> 0.49545 * log(100) + 4.63639
[1] 6.91802
```

Figure 43. R Code - 95% Confidence Interval Linear Model ECT

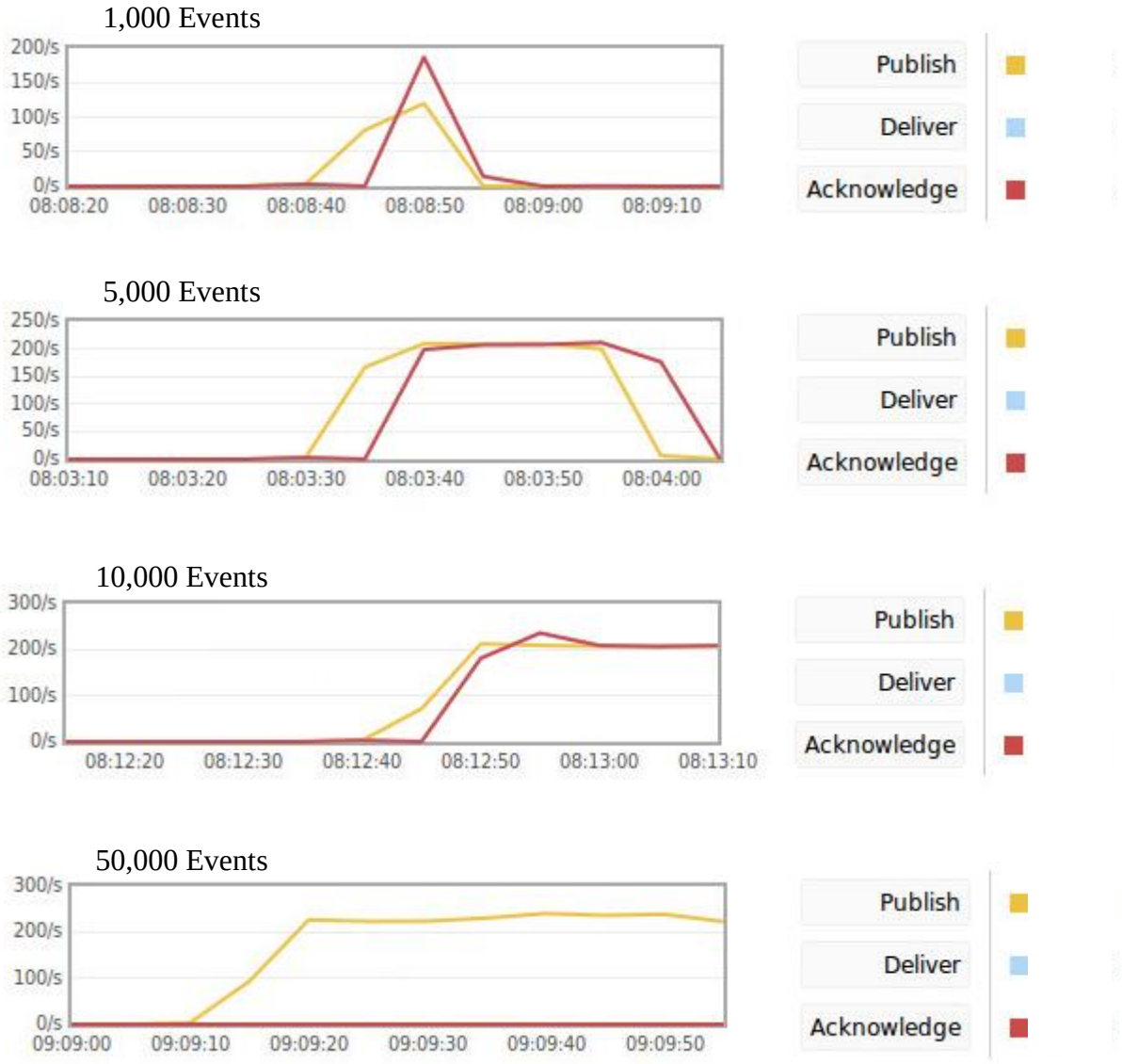


Figure 46. Experiment 1 - RabbitMQ Broker Message Rates

```
#### 10000
# read in raw files
r1_10000F1 = read.csv("C:\\Users\\mtodd\\Google Drive\\!Thesis!\\data\\FData\\10000
    ↪ _rF1_1.csv", header = FALSE)
r2_10000F1 = read.csv("C:\\Users\\mtodd\\Google Drive\\!Thesis!\\data\\FData\\10000
    ↪ _rF1_2.csv", header = FALSE)
r3_10000F1 = read.csv("C:\\Users\\mtodd\\Google Drive\\!Thesis!\\data\\FData\\10000
    ↪ _rF1_3.csv", header = FALSE)
<snipped>
r30_10000F1 = read.csv("C:\\Users\\mtodd\\Google Drive\\!Thesis!\\data\\FData\\10000
    ↪ _rF1_30.csv", header = FALSE)
# Event completion time
r1_10000F1$total <- rowSums(r1_10000F1[, c(2,3,4)])
r2_10000F1$total <- rowSums(r2_10000F1[, c(2,3,4)])
r3_10000F1$total <- rowSums(r3_10000F1[, c(2,3,4)])
<snipped>
r30_10000F1$total <- rowSums(r30_10000F1[, c(2,3,4)])
# Message delivery time
r1_10000F1$msgDeliverTime <- rowSums(r1_10000F1[, c(2,3)])
r2_10000F1$msgDeliverTime <- rowSums(r2_10000F1[, c(2,3)])
r3_10000F1$msgDeliverTime <- rowSums(r3_10000F1[, c(2,3)])
<snipped>
r30_10000F1$msgDeliverTime <- rowSums(r30_10000F1[, c(2,3)])

V2_10000F1_mean = c(mean(r1_10000F1$V2),mean(r2_10000F1$V2),mean(r3_10000F1$V2),mean(
    ↪ r4_10000F1$V2),mean(r5_10000F1$V2),mean(r6_10000F1$V2),mean(r7_10000F1$V2),
    ↪ mean(r8_10000F1$V2),mean(r9_10000F1$V2),mean(r10_10000F1$V2),mean(
    ↪ r11_10000F1$V2),mean(r12_10000F1$V2),mean(r13_10000F1$V2),mean(r14_10000F1$V2)
    ↪ ,mean(r15_10000F1$V2),mean(r16_10000F1$V2),mean(r17_10000F1$V2),mean(
    ↪ r18_10000F1$V2),mean(r19_10000F1$V2),mean(r20_10000F1$V2),mean(r21_10000F1$V2)
    ↪ ,mean(r22_10000F1$V2),mean(r23_10000F1$V2),mean(r24_10000F1$V2),mean(
    ↪ r25_10000F1$V2),mean(r26_10000F1$V2),mean(r27_10000F1$V2),mean(r28_10000F1$V2)
    ↪ ,mean(r29_10000F1$V2),mean(r30_10000F1$V2))

V3_10000F1_mean = c(mean(r1_10000F1$V3), mean(r2_10000F1$V3), mean(r3_10000F1$V3),
    ↪ mean(r4_10000F1$V3),mean(r5_10000F1$V3),mean(r6_10000F1$V3),mean(r7_10000F1$V3
    ↪ ),mean(r8_10000F1$V3),mean(r9_10000F1$V3),mean(r10_10000F1$V3),mean(
    ↪ r11_10000F1$V3),mean(r12_10000F1$V3),mean(r13_10000F1$V3),mean(r14_10000F1$V3)
    ↪ ,mean(r15_10000F1$V3),mean(r16_10000F1$V3),mean(r17_10000F1$V3),mean(
    ↪ r18_10000F1$V3),mean(r19_10000F1$V3),mean(r20_10000F1$V3),mean(r21_10000F1$V3)
    ↪ ,mean(r22_10000F1$V3),mean(r23_10000F1$V3),mean(r24_10000F1$V3),mean(
    ↪ r25_10000F1$V3),mean(r26_10000F1$V3),mean(r27_10000F1$V3),mean(r28_10000F1$V3)
    ↪ ,mean(r29_10000F1$V3),mean(r30_10000F1$V3))

V4_10000F1_mean = c(mean(r1_10000F1$V4),mean(r2_10000F1$V4),mean(r3_10000F1$V4),mean(
    ↪ r4_10000F1$V4),mean(r5_10000F1$V4),mean(r6_10000F1$V4),mean(r7_10000F1$V4),
    ↪ mean(r8_10000F1$V4),mean(r9_10000F1$V4),mean(r10_10000F1$V4),mean(
    ↪ r11_10000F1$V4),mean(r12_10000F1$V4),mean(r13_10000F1$V4),mean(r14_10000F1$V4)
    ↪ ,mean(r15_10000F1$V4),mean(r16_10000F1$V4),mean(r17_10000F1$V4),mean(
    ↪ r18_10000F1$V4),mean(r19_10000F1$V4),mean(r20_10000F1$V4),mean(r21_10000F1$V4)
    ↪ ,mean(r22_10000F1$V4),mean(r23_10000F1$V4),mean(r24_10000F1$V4),mean(
    ↪ r25_10000F1$V4),mean(r26_10000F1$V4),mean(r27_10000F1$V4),mean(r28_10000F1$V4)
    ↪ ,mean(r29_10000F1$V4),mean(r30_10000F1$V4))

V5_10000F1_mean = c(mean(r1_10000F1$V5),mean(r2_10000F1$V5),mean(r3_10000F1$V5),mean(
    ↪ r4_10000F1$V5),mean(r5_10000F1$V5),mean(r6_10000F1$V5),mean(r7_10000F1$V5),
    ↪ mean(r8_10000F1$V5),mean(r9_10000F1$V5),mean(r10_10000F1$V5),mean(
    ↪ r11_10000F1$V5),mean(r12_10000F1$V5),mean(r13_10000F1$V5),mean(r14_10000F1$V5)
    ↪ ,mean(r15_10000F1$V5),mean(r16_10000F1$V5),mean(r17_10000F1$V5),mean(
    ↪ r18_10000F1$V5),mean(r19_10000F1$V5),mean(r20_10000F1$V5),mean(r21_10000F1$V5)
    ↪ ,mean(r22_10000F1$V5),mean(r23_10000F1$V5),mean(r24_10000F1$V5),mean(
    ↪ r25_10000F1$V5),mean(r26_10000F1$V5),mean(r27_10000F1$V5),mean(r28_10000F1$V5)
    ↪ ,mean(r29_10000F1$V5),mean(r30_10000F1$V5))
```

Figure 44. Experiment 1 - R Code

```
total_10000F1_mean = c(mean(r1_10000F1$total),mean(r2_10000F1$total), <snipped> ,mean
    ↪ (r28_10000F1$total),mean(r29_10000F1$total),mean(r30_10000F1$total))

msgDeliverTime_10000F1_mean = c(mean(r1_10000F1$msgDeliverTime),mean(
    ↪ r2_10000F1$msgDeliverTime), <snipped> ,mean(r29_10000F1$msgDeliverTime),mean(
    ↪ r30_10000F1$msgDeliverTime))

#All data combined for each level
comb_10000F1 = data.frame(V2_10000F1_mean, V3_10000F1_mean, V4_10000F1_mean,
    ↪ V5_10000F1_mean, total_10000F1_mean, msgDeliverTime_10000F1_mean)
#Add headings to each col
names(comb_10000F1) <- c("Agent Message Time", "Consumer Message Time", "Controller
    ↪ Update Time", "Event Succes Rate", "Event Completion Time", "Message Delivery
    ↪ Time")
#Display pairs plots
pairs(comb_10000F1)
#View summary of data
summary(comb_10000F1)

#Print out table formatted for latex
paste(round(mean(comb_10000F1$`Event Completion Time`), digits=2) , " & " , round(sd(
    ↪ comb_10000F1$`Event Completion Time`), digits=2) , " & " , round(min(
    ↪ comb_10000F1$`Event Completion Time`), digits=2) , " & " , round(quart(
    ↪ comb_10000F1$`Event Completion Time`)[1], digits=2) , " & " , round(median(
    ↪ comb_10000F1$`Event Completion Time`), digits=2) , " & " , round(quart(
    ↪ comb_10000F1$`Event Completion Time`)[2], digits=2) , " & " , round(max(
    ↪ comb_10000F1$`Event Completion Time`), digits=2), " & ")
#power test used to determine sample size was sufficient
power.t.test(n = NULL, delta = 1, power = .90, sd = sd(comb_10000F1$`Agent Msg Time`)
    ↪ )
power.t.test(n = NULL, delta = 1, power = .90, sd = sd(comb_10000F1$`Consumer Msg
    ↪ Time`))
power.t.test(n = NULL, delta = 1, power = .90, sd = sd(comb_10000F1$`Controller
    ↪ Update Time`))

##### Combined Data
# Agent Message Time by number of eventts 95% Conf
boxplot(comb_10000F1$`Agent Message Time`,comb_50000F1$`Agent Message Time`, names=c(
    ↪ "10K", "50K"), main="Agent Message Time (ms) by \n Number of Events")
# Consumer Message Time by number of eventts 95% Conf
boxplot(comb_10000F1$`Consumer Message Time`,comb_50000F1$`Consumer Message Time`,
    ↪ names=c( "10K", "50K"), main="Consumer Message Time (ms) by \n Number of
    ↪ Events")
# Message Delivery Time by number of events
boxplot(comb_10000F1$`Message Delivery Time`,comb_50000F1$`Message Delivery Time`,
    ↪ names=c("10K", "50K"), main="Message Delivery Time (ms) by \n Number of Events
    ↪ ")
# Controller Update Time
boxplot(comb_10000F1$`Controller Update Time`,comb_50000F1$`Controller Update Time`,
    ↪ names=c("10K", "50K"), main="Controller Update Time (ms) by \n Number of
    ↪ Events")
# Event Completion Time
boxplot(comb_10000F1$`Event Completion Time`,comb_50000F1$`Event Completion Time`,
    ↪ names=c("10K", "50K"), main="Event Completion Time (ms) by \n Number of Events
    ↪ ")
# Elapse Time
boxplot(times10000/1000000000, times50000/1000000000, names=c("10K", "50K"), main="
    ↪ Elapse Time (sec) by \n Number of Events")

#### Timing data for tables in section 5.2
# time is in nanosecond, convert to seconds
summary(times10000/1000000000)
# mean msg rate
10000/mean(times10000/1000000000)
```

Figure 45. Experiment 1 - R Code Continued

Appendix D. Experiment 2 Graphs and Supporting R Code

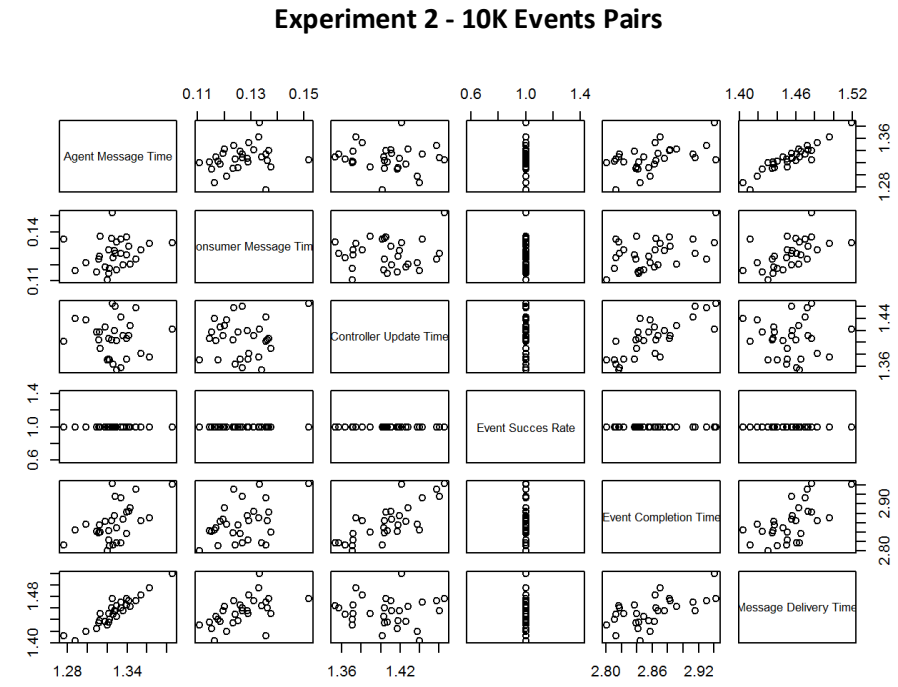


Figure 47. Experiment 2 - Pairs Plot 10K and 50K

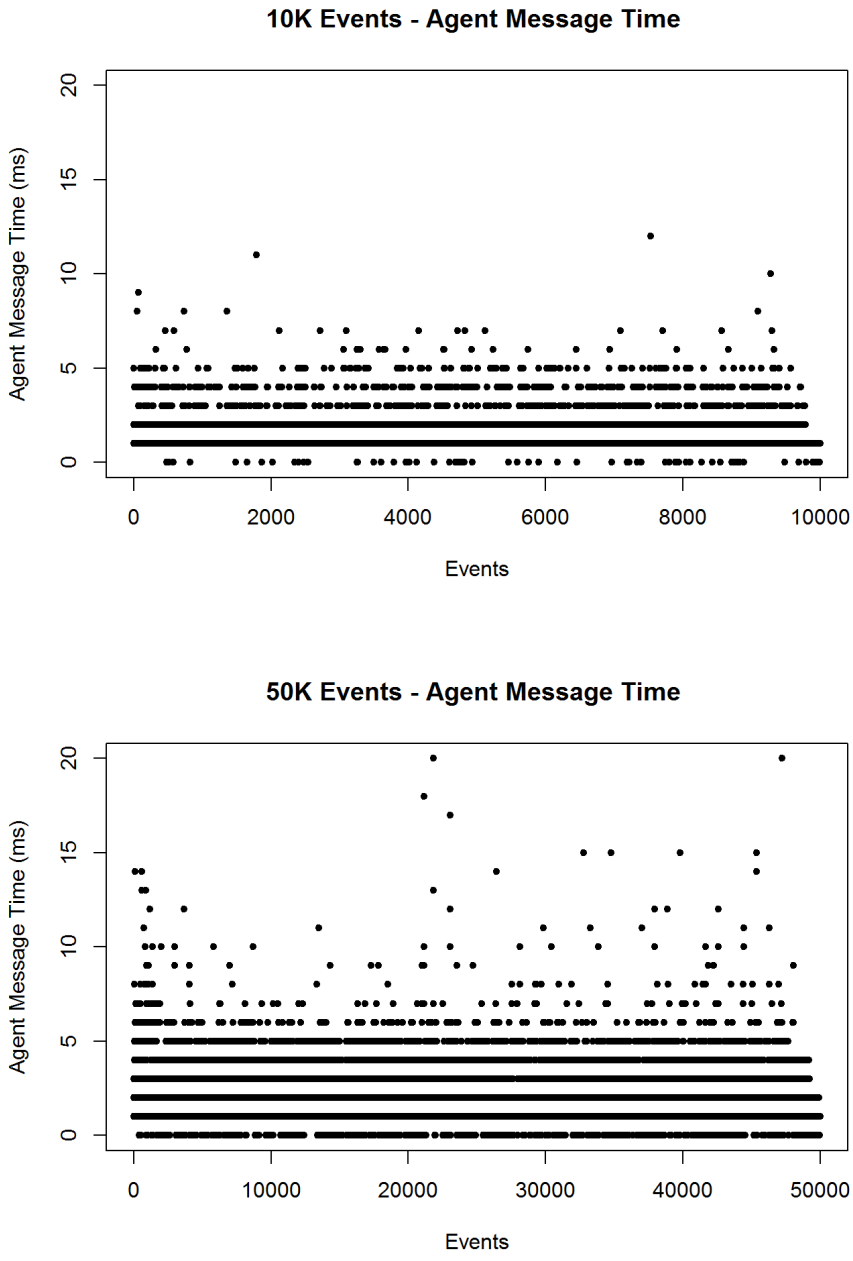


Figure 48. Experiment 1 - Agent Message Time Data Plot

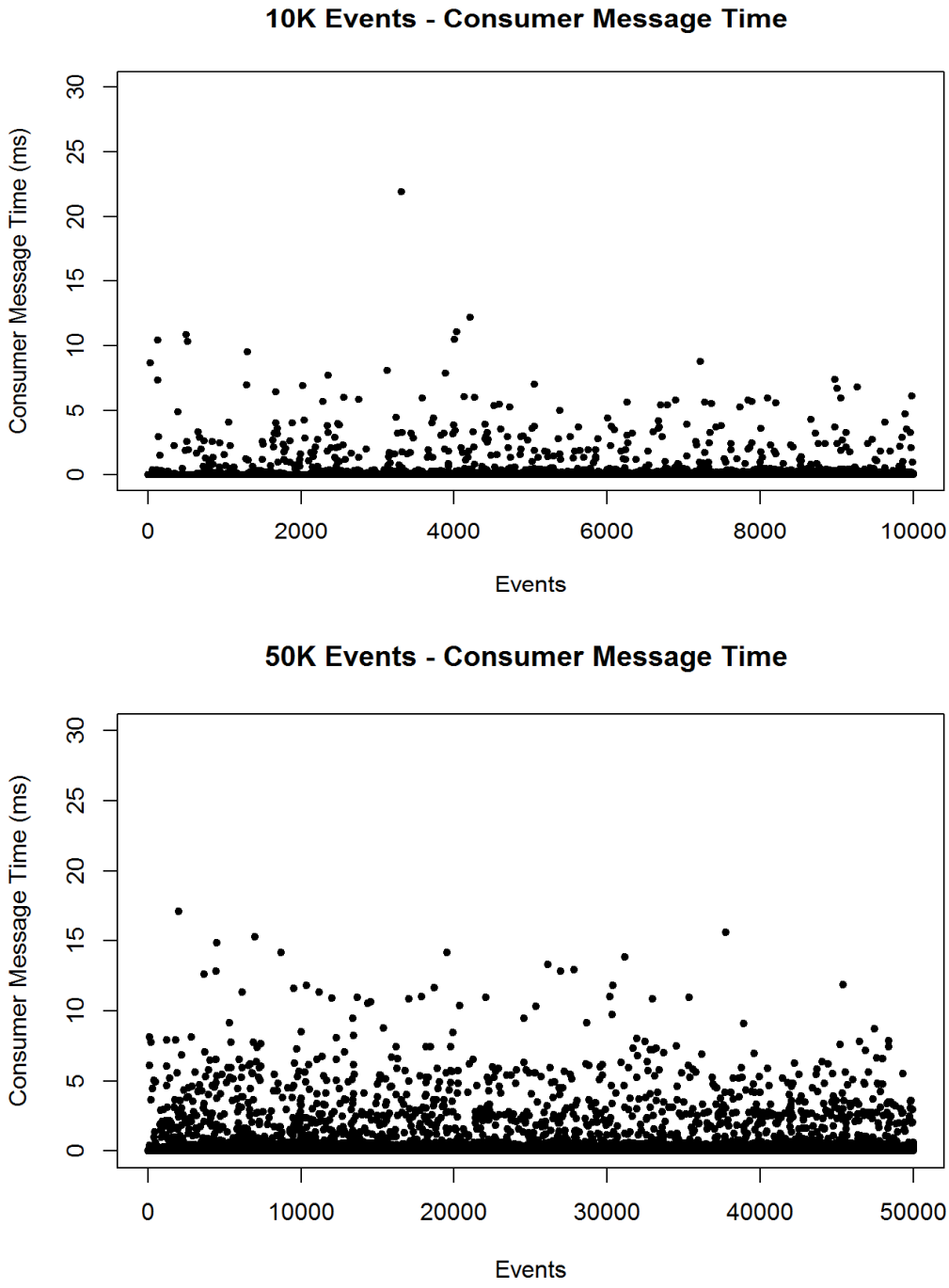


Figure 49. Experiment 1 - Consumer Message Time Data Plot

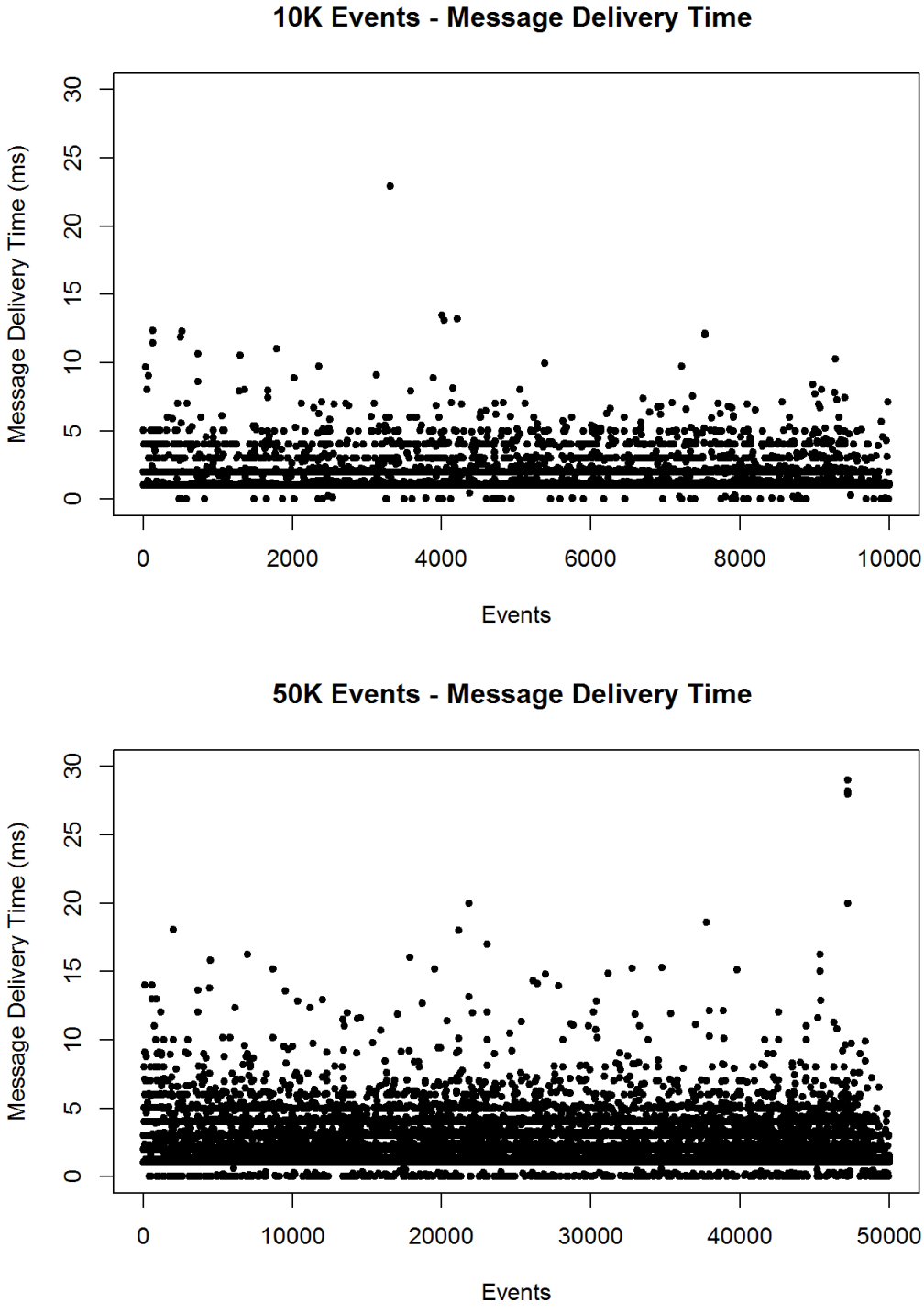
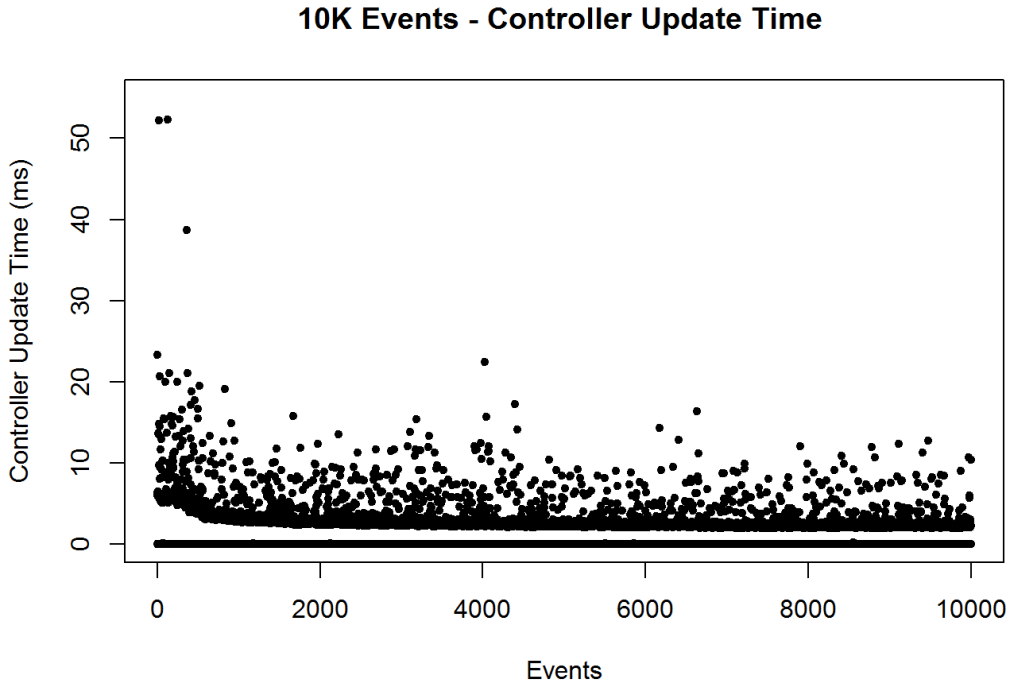


Figure 50. Experiment 1 - Message Delivery Time Data Plot



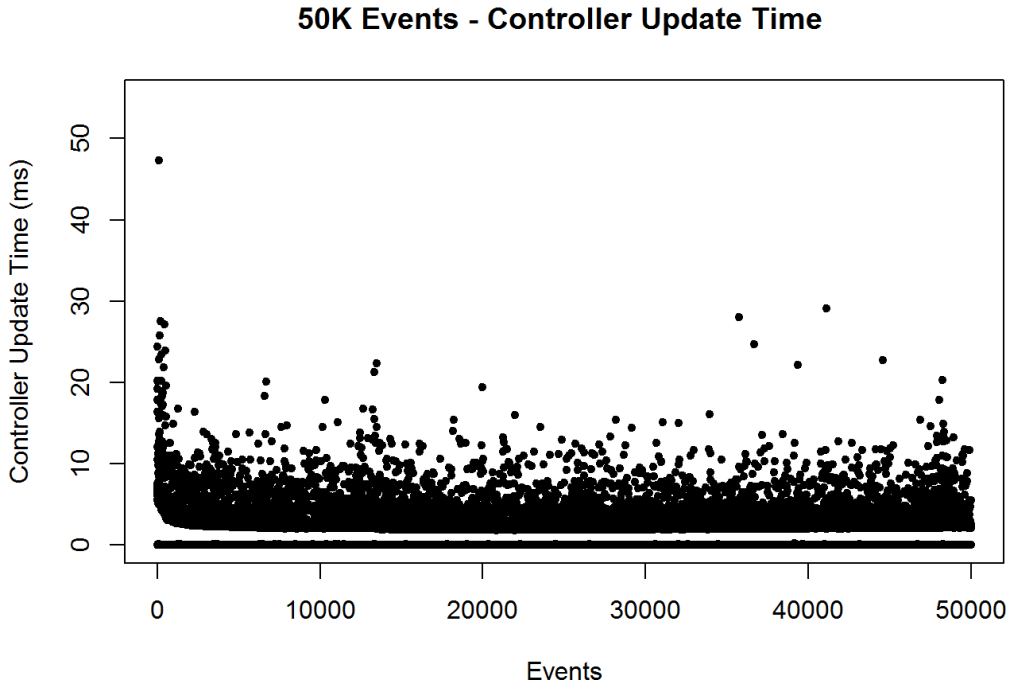
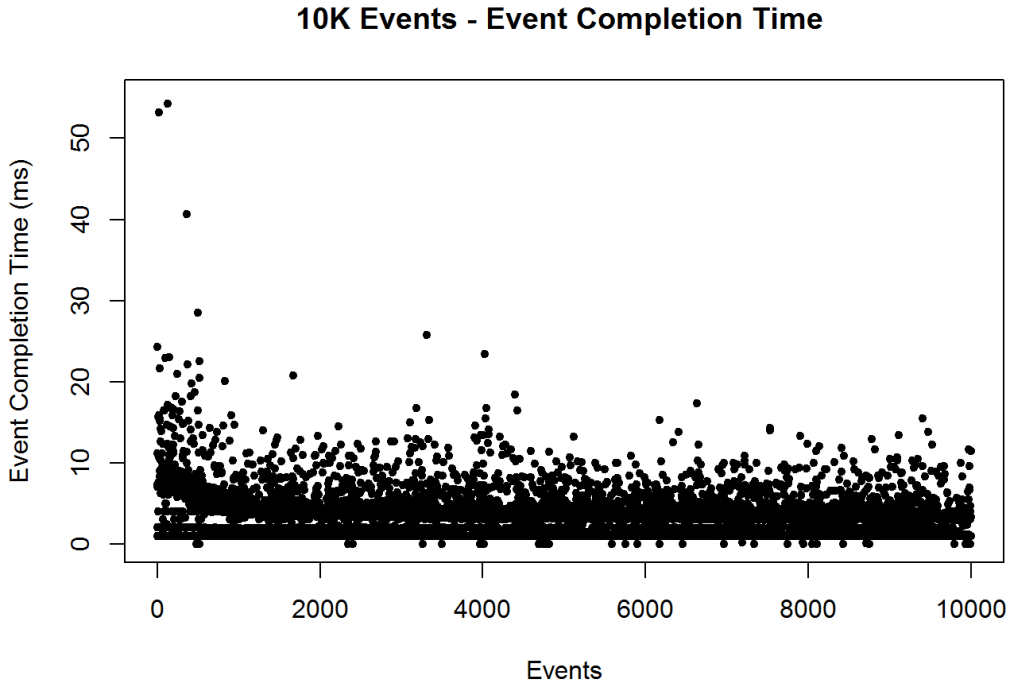


Figure 51. Experiment 1 - Controller Update Time Data Plot



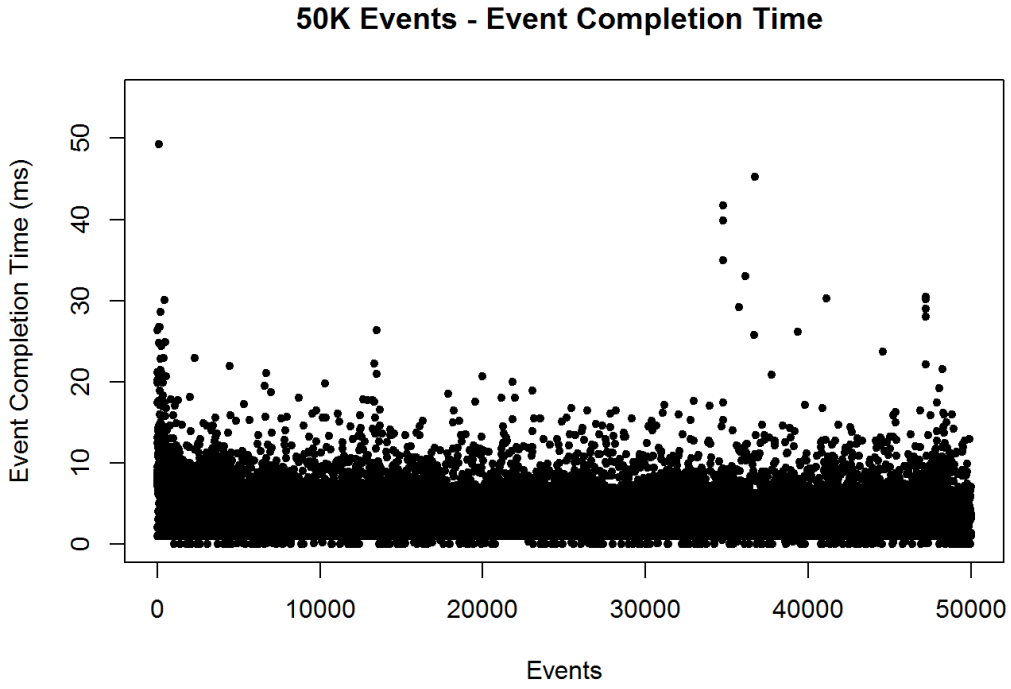
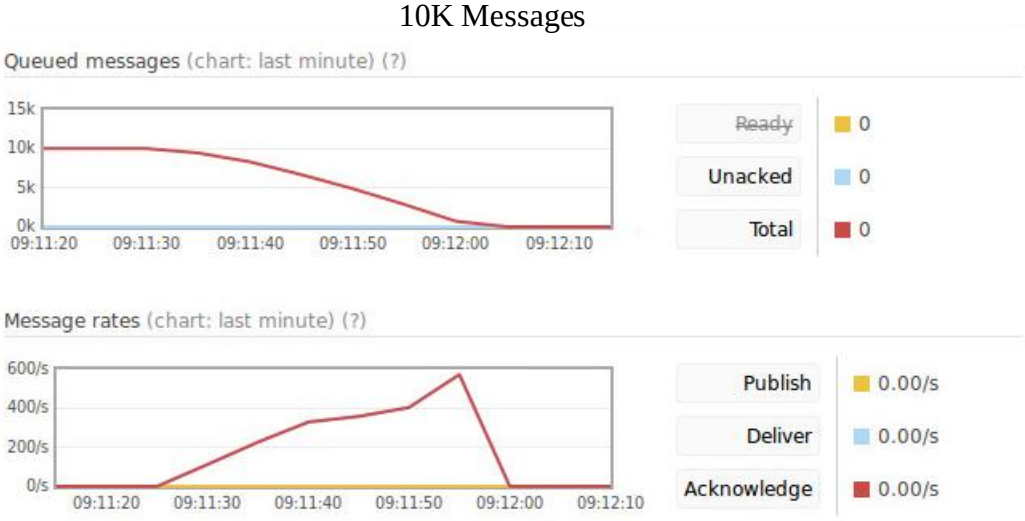


Figure 52. Experiment 1 - Event Completion Time Data Plot

10K Messages



50K Messages

Figure 53. Experiment 2 - RabbitMQ (Broker) Message Rates

```
#### 10000
# read in raw files
r1_10000F2 = read.csv("C:\\Users\\mtodd\\Google_Drive\\!Thesis!\\data\\FData\\10000
    ↪ _rF2_1.csv", header = FALSE)
r2_10000F2 = read.csv("C:\\Users\\mtodd\\Google_Drive\\!Thesis!\\data\\FData\\10000
    ↪ _rF2_2.csv", header = FALSE)
r3_10000F2 = read.csv("C:\\Users\\mtodd\\Google_Drive\\!Thesis!\\data\\FData\\10000
    ↪ _rF2_3.csv", header = FALSE)
<snipped>
r30_10000F2 = read.csv("C:\\Users\\mtodd\\Google_Drive\\!Thesis!\\data\\FData\\10000
    ↪ _rF2_30.csv", header = FALSE)
# Event completion time
r1_10000F2$total <- rowSums(r1_10000F2[, c(2,3,4)])
r2_10000F2$total <- rowSums(r2_10000F2[, c(2,3,4)])
r3_10000F2$total <- rowSums(r3_10000F2[, c(2,3,4)])
<snipped>
r30_10000F2$total <- rowSums(r30_10000F2[, c(2,3,4)])
# Message delivery time
r1_10000F2$msgDeliverTime <- rowSums(r1_10000F2[, c(2,3)])
r2_10000F2$msgDeliverTime <- rowSums(r2_10000F2[, c(2,3)])
r3_10000F2$msgDeliverTime <- rowSums(r3_10000F2[, c(2,3)])
<snipped>
r30_10000F2$msgDeliverTime <- rowSums(r30_10000F2[, c(2,3)])

V2_10000F2_mean = c(mean(r1_10000F2$V2[3001:length(r1_10000F2$V2)]),mean(
    ↪ r2_10000F2$V2[3001:length(r1_10000F2$V2)]),mean(r3_10000F2$V2[3001:length(
    ↪ r1_10000F2$V2)]), .<snipped>. ,mean(r30_10000F2$V2[3001:length(r1_10000F2$V2)
    ↪ ]))

V3_10000F2_mean = c(mean(r1_10000F2$V3[3001:length(r1_10000F2$V3)]), mean(
    ↪ r2_10000F2$V3[3001:length(r1_10000F2$V3)]), mean(r3_10000F2$V3[3001:length(
    ↪ r1_10000F2$V3)]), .<snipped>. ,mean(r29_10000F2$V3[3001:length(r1_10000F2$V3)
    ↪ ]),mean(r30_10000F2$V3[3001:length(r1_10000F2$V3)]))

V4_10000F2_mean = c(mean(r1_10000F2$V4[3001:length(r1_10000F2$V4)]), ... <snipped>...
    ↪  ,mean(r30_10000F2$V4[3001:length(r1_10000F2$V4)]))

V5_10000F2_mean = c(mean(r1_10000F2$V5[3001:length(r1_10000F2$V5)]), ... <snipped>...
    ↪  ,mean(r30_10000F2$V5[3001:length(r1_10000F2$V5)]))
```

Figure 54. Experiment 2 - R Code

```
total_10000F2_mean = c(mean(r1_10000F2$total[3001:length(r1_10000F2$V2)]),mean(
    ↪ r2_10000F2$total[3001:length(r1_10000F2$V2)]), <snipped> , mean(
    ↪ r29_10000F2$total[3001:length(r1_10000F2$V2)]),mean(r30_10000F2$total))

msgDeliverTime_10000F2_mean = c(mean(r1_10000F2$msgDeliverTime[3001:length(
    ↪ r1_10000F2$V2)]),mean(r2_10000F2$msgDeliverTime[3001:length(r1_10000F2$V2)]),
    ↪ <snipped> ,mean(r29_10000F2$msgDeliverTime[3001:length(r1_10000F2$V2)]),mean(
    ↪ r30_10000F2$msgDeliverTime[3001:length(r1_10000F2$V2)]))

#All data combined for each level
comb_10000F2 = data.frame(V2_10000F2_mean, V3_10000F2_mean, V4_10000F2_mean,
    ↪ V5_10000F2_mean, total_10000F2_mean, msgDeliverTime_10000F2_mean)
#Add headings to each col
names(comb_10000F2) <- c("Agent Message Time", "Consumer Message Time", "Controller
    ↪ Update Time", "Event Succes Rate", "Event Completion Time", "Message Delivery
    ↪ Time")
#Display pairs plots
pairs(comb_10000F2)
#View summary of data
summary(comb_10000F2)

#power test used to determine sample size was sufficient
power.t.test(n = NULL, delta = 1, power = .90, sd = sd(comb_10000F2$`Agent Msg Time`)
    ↪ )
power.t.test(n = NULL, delta = 1, power = .90, sd = sd(comb_10000F2$`Consumer Msg
    ↪ Time`))
power.t.test(n = NULL, delta = 1, power = .90, sd = sd(comb_10000F2$`Controller
    ↪ Update Time`))

##### Combined Data
# Agent Message Time by number of eventts 95% Conf
boxplot(comb_10000F2$`Agent Message Time`,comb_50000F2$`Agent Message Time`, names=c(
    ↪ "10K", "50K"), main="Agent Message Time (ms) by \n Number of Events")
# Consumer Message Time by number of eventts 95% Conf
boxplot(comb_10000F2$`Consumer Message Time`,comb_50000F2$`Consumer Message Time`,
    ↪ names=c( "10K", "50K"), main="Consumer Message Time (ms) by \n Number of
    ↪ Events")
# Message Delivery Time by number of events
boxplot(comb_10000F2$`Message Delivery Time`,comb_50000F2$`Message Delivery Time`,
    ↪ names=c("10K", "50K"), main="Message Delivery Time (ms) by \n Number of Events
    ↪ ")
# Controller Update Time
boxplot(comb_10000F2$`Controller Update Time`,comb_50000F2$`Controller Update Time`,
    ↪ names=c("10K", "50K"), main="Controller Update Time (ms) by \n Number of
    ↪ Events")
# Event Completion Time
boxplot(comb_10000F2$`Event Completion Time`,comb_50000F2$`Event Completion Time`,
    ↪ names=c("10K", "50K"), main="Event Completion Time (ms) by \n Number of Events
    ↪ ")
# Elapse Time
boxplot(times10000/1000000000, times50000/1000000000, names=c("10K", "50K"), main="
    ↪ Elapse Time (sec) by \n Number of Events")

#### Timing data for tables in section 5.2
# time is in nanosecond, convert to seconds
summary(times10000/1000000000)
# mean msg rate
10000/mean(times10000/1000000000)
```

Figure 55. Experiment 2 - R Code Continued

Bibliography

1. M. Casado, M. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking Control of the Enterprise. ACM SIGCOMM Computer Communications Review, pages 1-12, 2007.

2. R. Hand, M. Ton, and E. Keller, editors. *Active Security*, HotNets-XII, New York, NY, USA, 2013. ACM.

3. G. Abelar and D. Tesch. Role of CS-MARS in Your Network. Retrieved 28 December, 2015 from *http://www.ciscopress.com/articles/article.asp?p=664149*, 2006.

4. Symantec Corp. *Symantec Internet Security Threat Report Trends for 2015*, 2015.

5. C. Kaufman, R. Perlman, and M. Speciner. *Network Security: Private Communication in a Public World, Second Edition*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2nd edition edition, 2002.

6. P. Dowd and J. McHenry. Network Security: It's Time to Take it Seriously. *Computer*, 31(9):24–28, 1998.

7. B. Daya. Network security: History, Importance, and Future. *University of Florida Department of Electrical and Computer Engineering*, 2013.

8. G. Dowling and R. Staelin. A Model of Perceived Risk and Intended Risk-handling Activity. *Journal of Consumer Research*, 21(1):119–134, 1994.

9. J. Howard and T. Longstaff. A Common Language for Computer Security Incidents. *Sandia National Laboratories*, 1998.

10. DISA. Defense Information Systems Agency Host-based Security System. Retrieved 31 December, 2015 from *http://www.disa.mil/Cybersecurity/Network-Defense/HBSS*, 2015.

11. P. Kabiri. *Privacy, Intrusion Detection, and Response*. IGI Global, 2012.

12. A. Tanenbaum and D. Wetherall. *Computer Networks: Pearson New International Edition: University of Hertfordshire*. Pearson Higher Ed, forth edition edition, 2013.

13. R. Bace and P. Mell. NIST Special Publication 800-31: Intrusion Detection Systems, National Institute of Standards and Technology (NIST). Retrieved 6 March, 2015 from *http://csrc.nist.gov/publications/nistpubs/800-94/SP800-94.pdf*, 2001.

14. National Institute of Standards and Technology. 800-94, Guide to Intrusion Detection and Prevention Systems (IDPS). *Recommendations of the National Institute of Standards and Technology*, 2007.

15. M. Roesch. Snort: Lightweight Intrusion Detection for Networks. System Administrators Conf. Retrieved on 3 June from *http://static.usenix.org/publications /roesch/roesch.pdf*, 1999.

16. W. Lee and S. Stolfo. Data Mining Approaches for Intrusion Detection. Retrieved 5 March, 2015 from *Usenix Security.*, 1998.

17. D. Ferraiolo, J. Barkley, and D. Kuhn. Role-based Access Controls. In *Proceedings of 15th NIST-NCSC National Computer Security Conference*, volume 563. Baltimore, Maryland: NIST-NCSC, 1992. RBAC.

18. R. Sandhu, D. Ferraiolo, and R. Kuhn. The NIST Model for Role-based Access Control: Towards a Unified Standard. In *ACM workshop on Role-based access control*, volume 2000, 2000.

19. D. Ferraiolo, J. Barkley, and D. Kuhn. A Role-based Access Control Model and Reference Implementation within a Corporate Intranet. *ACM Transactions on Information and System Security (TISSEC)*, 2(1):34–64, 1999.

20. J. Postel. Transmission Control Protocol. Defence Advanced Research Techniques Office. Retrieved on 18 May 2015 from *https://www.ietf.org/rfc/rfc793.txt*, 1981.

21. T. Humernbrum, F. Glinka, and S. Gorlatch. A Northbound API for QOS Management in Real-time Interactive Applications on Software Defined Networks. *Journal of Communications*, 9(8):607 – 615, 2014.

22. M. Casado, T. Koponen, S. Shenker, and A. Tootoonchian. Fabric: A Retrospective on Evolving SDN. *Proceedings of the first workshop on Hot topics in software defined networks*, pages 85 – 90, 2012.

23. Open Networking Foundation. Software-defined Networking: The New Norm for Networks. *Software World*, (6):4, 2014.

24. K. Kirkpatrick. Software-defined networking. *Communications of the ACM*, 56(9):16 – 19, 2013.

25. B. Butler. Rackspace Leverages Openstack's Quantum Project. Network World. Retrieved 15 March, 2015 from *EBSCOHOST, ISBN: 0887-7661*, 2012.

26. Open Networking Foundation. About Open vSwitch. Retrieved on May 13, 2015 from *http://www.openvswitch.org*, 2014.

27. Cisco Systems. Cisco Nexus 5000 Series Architecture. Retrieved 9 October, 2015 from *http://www.cisco.com/c/en/us/products/collateral/switches/nexus-5020-switch/white_paper_c11-462176.pdf*, 2009.

28. IBM Corp. IBM Distributed Virtual Switch 5000v Quickstart Guide. Retrieved 9 October, 2015 from *http://www.redbooks.ibm.com/redbooks/pdfs/sg248115.pdf*, 2014.

29. ONF. What is ONF?. Retrieved 9 October, 2015 from *https://www.opennetworking.org/about/onf-overview*, 2014.

30. N. Feamster, J. Rexford, and E. Zegura. The Road to SDN: An Intellectual History of Programmable Networks. *ACM SIGCOMM Computer Communication Review*, 44(2):87–98, 2014. ID: 000334996900014.

31. J. Michael, Z. Thomas, H. Tobias, T. Phuoc, and K. Wolfgang. Interfaces, Attributes, and Use Cases: A Compass for SDN. *IEEE Communications Magazine*, 52(6):210–217, 06 2014.

32. Floodlight-Project. Floodlight Documentation. Retrieved 7 October, 2015 from *http://docs.projectfloodlight.org/display/floodlightcontroller/For+Developers*, 2015.

33. OpenStack Networking. Networking Capabilities. Retrieved on 12 May, 2015 from *http://www.openstack.org/software/openstack-networking/*, 2015.

34. S. Scott-Hayward, G. O'Callaghan, and S. Sezer. SDN Security: A Survey. In *Future Networks and Services (SDN4FNS), 2013 IEEE SDN for*, pages 1–7, Nov 2013.

35. R. Kloti, V. Kotronis, and P. Smith. Openflow: A Security Analysis. In *Network Protocols (ICNP), 2013 21st IEEE International Conference on*, pages 1–6, Oct 2013.

36. P. Biondi. Scapy Packet Manipulation. Retrieved 5 March, 2015 from *http://www.secdev.org/projects/scapy/*, 2010.

37. J. O'Hara. Toward a Commodity Enterprise Middleware. *Queue*, 5(4):48–55, 2007.

38. International Organization for Standards. ISO/IEC 19464:2014 Information Technology Advanced Message Queuing Protocol (AMQP) v1.0 Specification, 2014.

39. OASIS. Advanced Message Queuing Protocol. Retrieved 5 March, 2015 from *https://www.amqp.org*, 2015.

40. S. Vinoski. Advanced Message Queuing Protocol. *Internet Computing, IEEE*, 10(6):87–89, Nov 2006.

41. Pivotal Software. RabbitMQ Features: What can RabbitMQ do for you? Retrieved 5 June, 2015 from *http://www.rabbitmq.com/features.html*, 2015.

42. S. Boschi and G. Santomaggio. *RabbitMQ Cookbook*. Packt Publishing Ltd, 2013.

43. M. Rostanski, K. Grochla, and A. Seman. Evaluation of Highly Available and Fault-tolerant Middleware clustered architectures using rabbitmq. In *Computer Science and Information Systems (FedCSIS), 2014 Federated Conference on*, pages 879–884. IEEE, 2014.

44. Symantech Corporation. Symantec Scan Engine 5.2 Manual. Retrieved 5 January, 2016 from *ftp://ftp.symantec.com/ symantec-scan-engine/5.2/manuals/developers-guide.pdf*, 2008.

45. A. Mirzazhanov. APG Manual Ubuntu 14.04. Retrieved 6 January, 2016 from *https://help.ubuntu.com/community/StrongPasswords*, 2015.

46. Big Switch Networks. Host-based Security Systems Guides. Retrieved 2 October, 2015 from *https://github.com/floodlight*, 2015.

47. Pivotal Software. RabbitMQ Java Client Library. Retrieved 2 October, 2015 from *https://www.rabbitmq.com/java-client.html*, 2015.

48. Floodlight-Project. Firewall REST API. Retrieved 7 October, 2015 from *https://floodlight.atlassian.net/wiki/display/floodlightcontroller/Firewall+REST+API*, 2015.

49. M. J. Crawley. *Statistics: An Introduction using R.* John Wiley & Sons, 2014.

50. R Core Team. R language definition, 2000.

51. Hobbit. Netcat TCP and UDP application. Avian Research. Retrieved on 1 January May from *https://www.ietf.org/rfc/rfc793.txt*, 1995.

52. Hewlett-Packard. Quickspecs HP 5900 Switch Series. Retrieved 29 October, 2015 from *http://h18000.www1.hp.com/products/quickspecs/14252 na/14252 na.pdf*, 2015.

53. International Secure Systems Labs. Anubis Malware Analysis of Unknown Binaries. Retrieved 23 December, 2015 from *https://anubis.iseclab.org/*, 2015.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
24–03–2016	Master's Thesis	Aug 2014 — Mar 2016

4. TITLE AND SUBTITLE

Dynamic Network Security Control Using Software Defined Networking

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Todd, Michael, C., Capt, USAF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENG-MS-16-M-049

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Intentionally Left Blank

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A: APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This thesis develops and implements a process to rapidly respond to host level security events using a host agent, Software Defined Networking and OpenFlow updates, role based flow classes, and Advanced Messaging Queuing Protocol to automatically update configuration of switching devices and block malicious traffic. Results show flow table updates are made for all tested levels in less than 5.27 milliseconds and event completion time increased with treatment level as expected. As the number of events increases from 1,000 to 50,000, the design scales logarithmically caused mainly by message delivery time. Event processing throughput is limited primarily by the message rate of the agent (40 msg./sec.). Additionally, the maximum effective consume rate for the controller indicates this design is capable of supporting up to 380 hosts at one msg./sec. Finally, every event triggered is successfully processed for both experiments resulting in a 100% event success rate.

15. SUBJECT TERMS

SDN, OpenFlow, Security, AMQP, Agent

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dr. Barry E. Mullins (ENG)
U	U	U	U	113	19b. TELEPHONE NUMBER *(include area code)* (937) 255-3636 x7979 Barry.Mullins@afit.edu